COO-3077-157

Courant Mathematics and
Computing Laboratory

U.S. Department of Energy

The Structure of the Puma Computer System

Overview and the Central Processor

Ralph Grishman

U.S. Department of Energy Report

Prepared under Contract EY-76-C-02-3077
with the Office of Energy Research

Mathematics and Computing
November 1978

New York University

UNCLASSIFIED

Courant Mathematics and Computing Laboratory

New York University

Mathematics and Computing COO-3077-157

THE STRUCTURE OF THE PUMA COMPUTER SYSTEM

Overview and the Central Processor

Ralph Grishman

November 1978

U. S. Department of Energy

Contract EY-76-C-02-3077

UNCLASSIFIED

CONTENTS

		Page
1.	Project objectives	1
2.	A chronology	3
3.	System structure	5
4.	Central processor structure	7
5.	Microprogramming the PUMA	29
6.	The microprogram for CDC 6600 emulation	46

1. Project objectives

Because of the rapidly decreasing cost of digital electronics, it has become possible in the last few years to assemble substantial digital computing elements at a modest component cost. In 1974, in response to these continuing hardware developments, we considered several possible digital design projects; we felt that such a project could develop a valuable expertise at the Courant Computing Laboratory and produce a useful product. We selected as our project an emulator for the Control Data 6600 central processor.

This choice was based on several considerations. Emulation of an existing machine would make a large amount of software immediately available. The CDC 6600, in addition to being available at the Laboratory, has a simple instruction set which is therefore easy to emulate. Finally, the projected materials cost for the project was relatively low. For a machine with perhaps half the computing power of a 6600, the estimated cost (exclusive of peripherals) was roughly $100,000 —— about 2 or 3% of the original price of a 6600. Because of further decreases in component prices since then, the total cost for components and wiring of circuit boards proved to be somewhat lower (in the vicinity of $65,000 for a machine with one million bytes of memory).

Low component costs are not very helpful unless manpower expenditures can be kept comparably low. This can be done only through a highly automated system for design, assembly, and testing. A large part of the project effort has been invested in developing such a system. Using this system, all circuits are thoroughly simulated prior to assembly; wiring lists are prepared by machine and circuit boards are wired automatically; assembled boards are tested by comparison with a simulated circuit. This system is described in detail in a separate report.

The system we have built has been dubbed the PUMA Computer System. Officially, PUMA is an acronym for Processing Unit with Microprogrammed Arithmetic, but the name is also intended to convey the grace of its design and the power of the system. As of the issuance of this report (Nov. 1978), PUMA serial number 1 has been running programs on a limited production basis for about four months and construction has begun on serial number 2. A briefy chronology of the PUMA project is given in Chapter 2.

This report focusses primarily on the structure of the PUMA system at the register and microprogram level. Chapter 3 briefly describes the overall system structure, while Chapter 4 goes into much greater detail regarding the central processor. Chapter 5 describes the microprogramming language of the PUMA, and can serve as an introduction for the microprogrammer. Finally, Chapter 6 presents the microprogram for emulation of the Control Data 6600.

2. A Chronology

1974

July 1	brief initial proposal circulated
Oct. 14	more detailed proposal, with some microcode sequences, prepared
Nov. 8	proposal presented at Computer Science Seminar

1975

through May	microprogram revised and completed
	4-bit ECL slice, used as decimal counter, wired
June	J. Fisher joins project
	chip-level design of arithmetic unit (AU)
	circuit simulator coded and tested
July	ECL slice tested at 30 MHz
	testing of AU design with simulator begins
Fall	debugging of AU design and microprogram
	chip placement for AU
	design of board testing system
Nov. 24	level-converter board for test system sent for wiring

1976

Jan. 23	testing of AU design and microprogram completed
	AU board sent for wiring; chips ordered
Spring	software for board-testing system developed (Generale)
	board testing system tested
May 19	last chips for AU received
June-Sept.	AU tested
Aug. 3	instruction unit sent for wiring
Fall	slow control unit designed
	memory and memory switch designed (Bianchini)

1977

Jan. 17	slow control unit sent for wiring
Feb.-March	software for testing processor ("PUMA utility") developed on H-316 (Generale)

	small memory (16 K words max) and memory switch built (Bianchini et al.)
April	system integrated
	initial, very low speed processor tests
Summer	microprogram debugged
	POS (Puma Operating System) installed on H-316 (Kenner)
October	large memory (131 K words) operational (Bianchini)
	fast control unit simulation begins (Grad)
Dec. 15	fast control unit sent for wiring
1978	
January	PDP-11/34 installed
	PUMA Utility written on PDP-11
February	PROMs for fast control unit programmed (Grad)
March	PUMA running with fast control unit
May	Puma Operating System installed on PDP-11 (Kenner)

3. System structure

In our system, as in the Control Data 6600, separate processors have been provided for executing user programs and managing peripherals. We are keeping open the option of duplicating the 6600 architecture, with its set of ten peripheral processors.* However, our initial approach has been to use a single commercially available minicomputer for controlling the peripherals.

This approach has a number of disadvantages. The largest is that all operating system functions currently performed by peripheral processors will have to be redesigned and recoded to run on the minicomputer or the central processor. Another possible problem is that a single minicomputer of modest price (about the same cost as the central processor we have built) may not be able to transmit data at a rate sufficient to keep the central processor busy.

There are several countervailing advantages to our approach. Most obviously, it means that we have to design and build less in order to obtain a working computer system. Moreover, the motivation — in purely hardware terms — is less strong for building peripheral processors than for building a central processor. High-speed, large word size scientific processors are still quite expensive, so the potential saving from a new, simple design is large. The slower, small-word-size machines suitable as peripheral processors, on the other hand, are now being mass-produced and have become quite inexpensive; the potential saving there is much smaller. Furthermore, the market for minicomputer-compatible peripheral controllers is very competitive, and consequently these controllers have become quite inexpensive. (In contrast, controllers compatible with Control Data peripheral processors are single-sourced and in some cases would be as expensive as the central processor and memory.)

* Work on a set of compatible peripheral processors is currently underway at Brookhaven National Laboratories.

The gross structure of our computer system is shown in Figure 1. A central processor (PUMA) and a peripheral-managing minicomputer (MINI) can each address all of a large central memory (CM). All data transfers to and from peripherals go through the minicomputer into central memory.

References to central memory go through a memory switch (MS). The memory itself has 60 data bits plus 4 parity bits per word. The path to PUMA is 60 bits wide, to the minicomputer 16 bits wide; the memory switch includes registers for assembling four 16-bit words into a 60-bit word when writing memory, and correspondingly disassembling words when reading. In addition, the memory switch provides direct data paths between PUMA and the minicomputer; these can be used by the minicomputer to control PUMA (start it, stop it, or switch tasks) and to run diagnostic tests of PUMA.

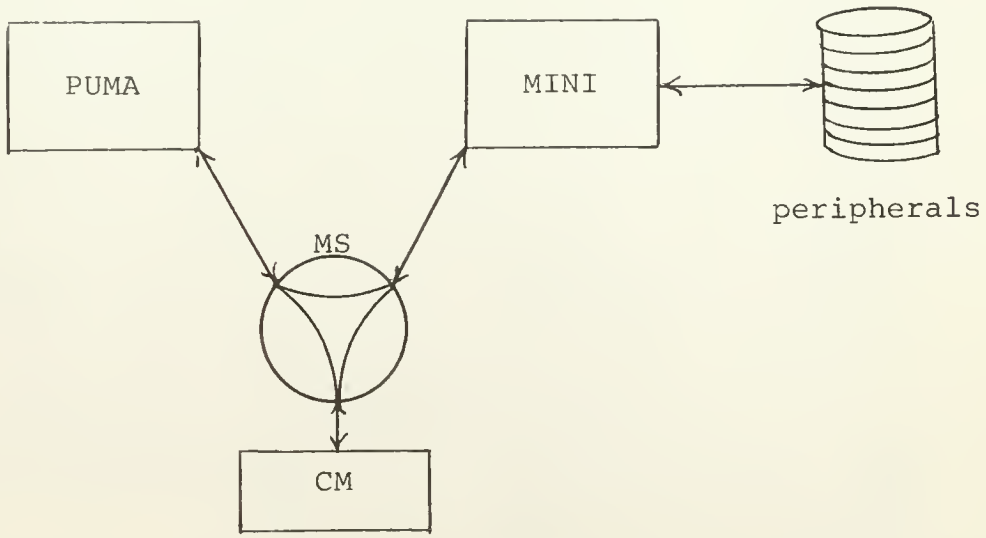

Figure 1. Gross Structure of the PUMA Computer System

4. Central processor structure

As we noted earlier, this project was prompted by developments in digital integrated circuit technology. The specific family of circuits we have used in our central processor is 10,000 series ECL (emitter-coupled logic). This family possesses two characteristics essential to our project: low gate propagation delay and availability of medium-scale integration chips.

The propagation delay of discrete gates in this series is 1.5 - 2.5 ns; gates included in larger functions have effective delay times closer to 1 ns. This represents a three- or four-fold increase in speed over the CDC 6600. Although somewhat faster logic families are available (with delay times for discrete gates below 1 ns) they provide a smaller variety of functions and are more difficult to interconnect (because of the high signal frequency).

The ECL 10,000 series includes a number of logic functions in 4-bit wide slices. Among the chips available are: a 4-bit shift register, a 4-bit counter, a 4-bit arithmetic-logic unit, and a 4-bit, 16-word register file. This level of integration makes it possible, for example, to build a 60-bit carry-look-ahead adder from 20 integrated circuit chips.

Because of the faster circuitry, we believed that we could build a machine nearly as fast as the 6600 using a very simple design with a minimum of parallelism. The design we have adopted, requiring fewer than 700 chips for the entire central processor, is described below.

The PUMA central processor is composed of three *units*: an arithmetic unit, an instruction unit, and a control unit (Figure 2). The arithmetic unit contains all the user programmable registers (A, B, and X registers), a number of registers for holding intermediate results in instruction interpretation, and the hardware for performing addition, subtraction, and Boolean operations on data. The instruction unit contains

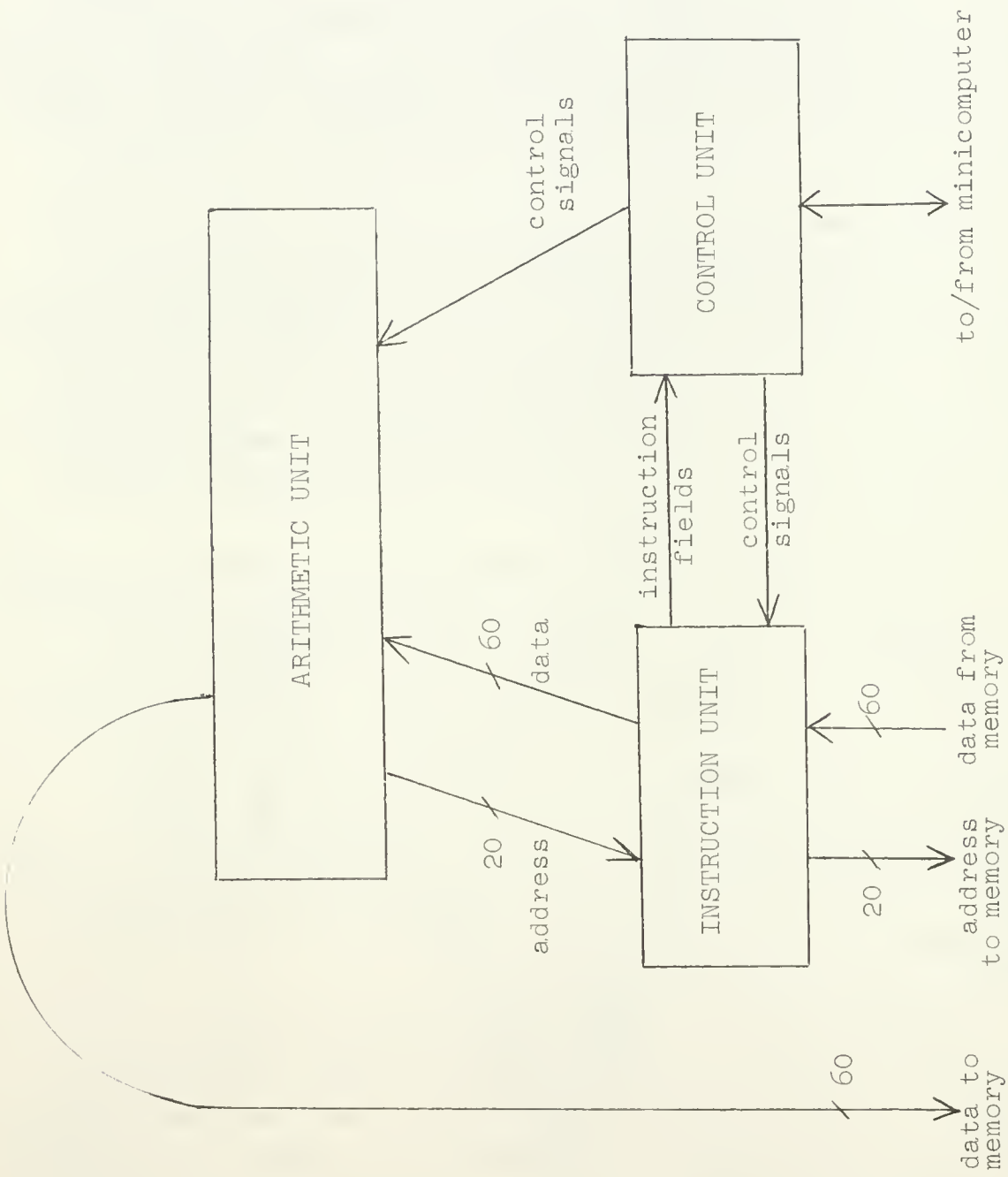

Figure 2. Units and main data paths of the PUMA central processor.

the logic for decoding instructions. In addition, it contains the P register (program counter), and a register for holding the next instruction word, which is fetched while the current instruction word is being executed. The control unit, of horizontal microprogrammed design, generates all control signals for the arithmetic and instruction units.

Figure 2 also shows the main data paths of PUMA. Data from memory (a 60-bit path) comes to the instruction unit. If it represents an instruction to be executed, it is held in the instruction unit; if it represents data to be loaded into an arithmetic register, it is sent through the instruction unit to the arithmetic unit. (Although this does add a few nanoseconds to the register load time, it avoids the need for a separate path from memory to the arithmetic unit. Because of the wide data path (60 bits), minimizing the number of such paths has been an important design consideration.) Data to be stored in memory originates only in the arithmetic unit, so the data path goes from the arithmetic unit to memory. Memory addresses may be generated either in the arithmetic unit (for register loads and stores) or in the instruction unit (for instruction fetches), so the address path goes from the arithmetic unit to the instruction unit and from the instruction unit to the memory. Instruction fields are transmitted from the instruction unit to the arithmetic unit (over the 60-bit data path) and to the control unit.

4.1 The Arithmetic Unit

The arithmetic unit is divided into two sections, a main arithmetic unit and an exponent arithmetic unit. The main unit is 60 bits wide, the exponent unit 12 bits wide. The exponent unit is used for counting and, as its name indicates, for exponent calculations in floating-point instructions.

Figure 3 shows the stucture of the main arithmetic unit. In the narrative which follows we shall slowly work our way from left to right in the figure.

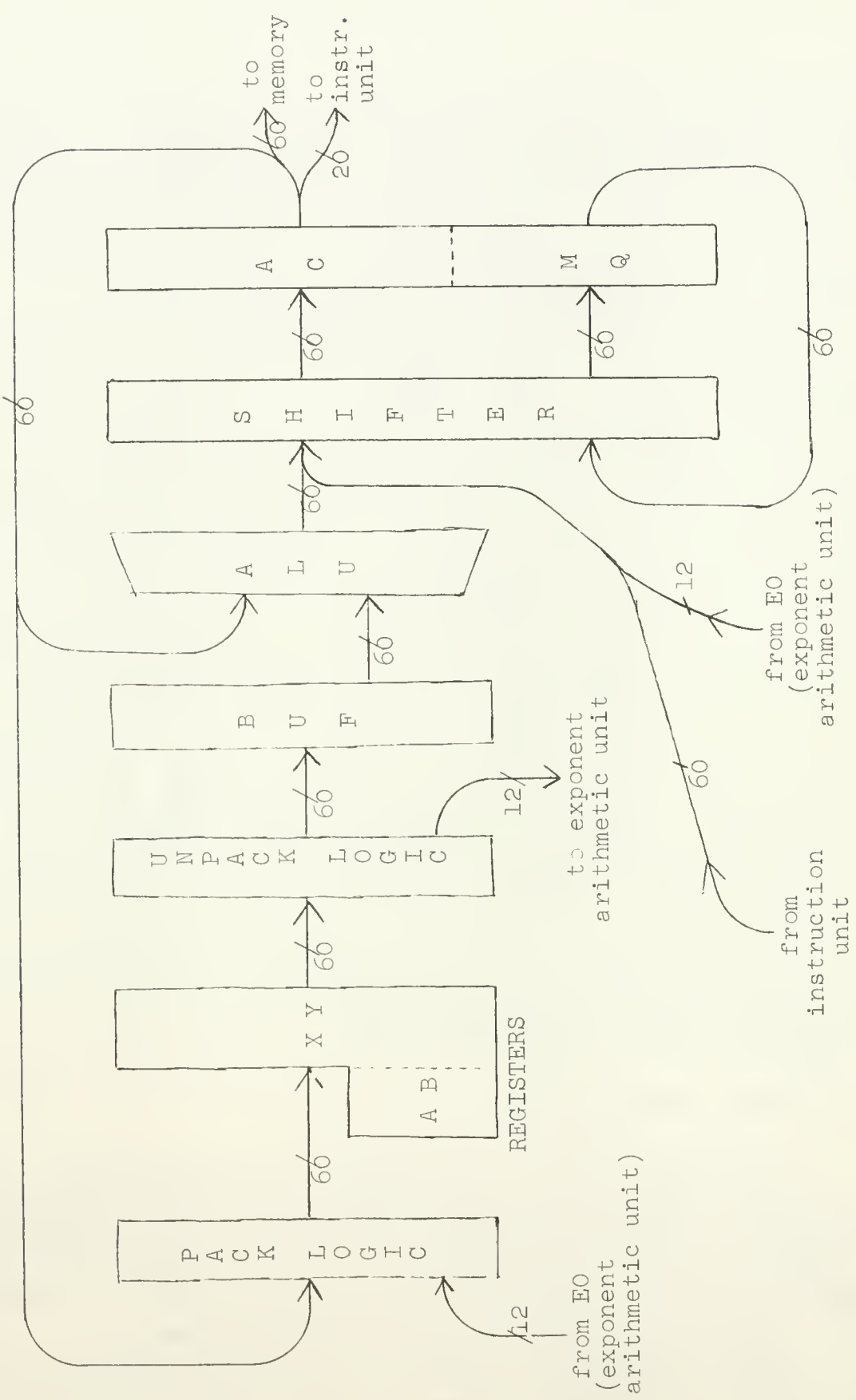

Figure 3. PUMA central processor: main arithmetic unit.

Near the left of the figure are the A, B, X, and Y registers. Each is a set of eight registers, named A0 through A7, B0 through B7, X0 through X7, and Y0 through Y7. The A and B registers are 20 bits wide, the X and Y registers 60 bits. The A, B, and X registers correspond to the user-programmable registers of the same names on the CDC 6600; the Y registers hold intermediate results during instruction interpretation. (The particular register configuration was dictated in part by the 16 word X 4 bit size of the ECL register chip. Thus, compatability with the 6600 required A and B registers of only 18 bits, but the 19^{th} and 20^{th} bits were "free". Similarly, we probably could have managed with just eight 60-bit registers, but eight more were available, so we wrote the microprogram to make good use of them.)

The output of the register array is fed to the unpack logic. The unpack logic is a hardware implementation of the 6600 unpack instruction, which separates a 6600-format floating point number into a coefficient (sign-extended to a 60-bit number) and an exponent. The exponent is sent to the exponent arithmetic unit. Operation of the unpack logic is controlled by a signal from the control unit — if not selected, the output of the registers passes through unchanged.

The output of the unpack logic goes to a 60-bit buffer register BUF. The presence of a buffer register between the register array and the arithmetic and shifting logic permits a limited degree of parallelism in the main arithmetic unit: an arithmetic operation can be performed on one operand while the next operand is being fetched from the register array or the previous result is being stored in the register array. (If the buffer register were not present, a number could be read out of the register array and used as an operand in an arithmetic operation in a single clock cycle. This would mean that certain sequences of operations could be performed in fewer clock cycles. However, each cycle would have to be considerably longer, since it would have to allow for both

register array access time and arithmetic operation time. In consequence, the effective speed of the machine would probably be significantly reduced.)

The box labeled ALU in the figure is a row of fifteen ALU (arithmetic and logic unit) chips, each four bits wide. One operand of the ALU comes from the BUF register, the other from the AC register. The ALU can perform addition, subtraction, and all sixteen Boolean operations.

For addition and subtraction, the ALU acts as a *subtractive* adder. A subtractive adder may best be though of as a circuit which first complements both operands, then adds them in the way in which you are familiar, and then complements the result. For two's complement arithmetic, a subtractive adder works just like a normal adder; for one's complement arithmetic, it has the feature tnat the sum of a number and its complement is plus zero (rather than minus zero for a normal adder). It is important to keep in mind, however, that when carry in, carry out, group generate, and group propagate are discussed, they are with respect to an addition being performed on the complements of the operands.

Several variations are possible on the addition and subtraction operations. The normal mode of addition is one's complement addition (carry out of the high-order bit is propagated end-around). One variant (the "no propagate" option), disables the end-around carry, so there is no carry into the low-order bit (this has the effect of a *normal* two's complement adder *with* a carry forced into the low-order bit). Another variant (the "generate" option), disables the end-around carry but forces a carry into the low-order bit (this has the effect of a *normal* two's complement adder *without* a carry into the low-order bit). A third variant is designed to facilitate 120-bit one's complement addition. Whenever an addition or subtraction is performed, the 60-bit group generate and propagate values can be saved in two flip-flops SAVEP and SAVEG. These flip-flops can then be used to control the end-around carry in a subsequent arithmetic operation, as follows:

let C_{60} be the carry out of the high order bit; then, under this variant, the carry brought around to the low-order bit will be

$$(C_{60} \wedge \text{SAVEP}) \vee \text{SAVEG}$$

So, to compute the low-order half of a 120-bit sum, one first saves the P and G values for the high-order halves of the operands and then adds the low-order halves using the saved P and G to control the end-around carry. The high-order half of the sum is similarly obtained.

One further variant performs 18-bit arithmetic. In this mode, the low 18 bits of AC and BUF are added (or subtracted) in what is effectively an 18-bit adder, and the sign bit of the result is extended to the high 42 bits. This mode matches the "increment unit" arithmetic of the 6600.

The output of the ALU is wire-ored with data coming from the instruction unit and from register E0 of the exponent arithmetic unit. These signals, together with the output of 60-bit register MQ, are fed into a 120-bit shifter. This is a combinatorial shifter made out of 4-way multiplexers. It can send the data straight through (no shift) or shift it to the right by 4, 16, or 60 bits. The 4 and 16-bit shifts can be either circular or arithmetic (high bit of ALU output extended to fill vacated positions); the 60-bit shift is always circular (interchanges ALU and MQ outputs). (Considerable thought during machine design was given to the selection of the best combination of shifts. An increase in the number of different shifts would have considerably speeded up the machine but was deemed too great an increase in the processor size. For example, a shifter which could shift any number of positions from 0 to 63 would have increased the size of the shifter from 60 to 180 chips, nearly a 20% increase in total processor size.)

The output of the shifter feeds a pair of 60-bit registers, AC and MQ. Each of these registers is itself a parallel entry shift register, so each, in addition to loading the output of

the shifter, can shift its data one bit to the left or right. On a one-bit left shift, the low bit of the AC receives the high bit of the MQ; the low bit of the MQ receives either the high bit of the AC or a constant 0 or 1, depending on select lines from the control unit. On a one-bit right shift, the high bit of the AC receives either the low bit of the MQ (circular shift) or remains unchanged (arithmetic shift); the high bit of the MQ receives the low bit of the AC.

The AC is the central register of the processing unit. The output of the AC is directly connected by cables to the memory switch; data to be written into memory is first loaded into the AC. By appropriately setting the memory switch, the minicomputer can read these signals and hence read the AC. This feature makes it easy to trace the processor during single-stepped execution. The low 20 bits of the AC are also routed to the instruction unit and are used to transmit the memory address for register loads and stores. Finally, the AC feeds the register array, so data to be stored in the register array must first go into the AC. On the path from the AC to the register array is the pack logic. This performs the inverse function from the unpack logic: it combines a coefficient in AC and a signed exponent in register E0 into a floating point number. This operation is performed only if selected by the control unit (using the same select line which controls the unpack logic).

Figure 4 shows the structure of the exponent arithmetic unit. The unit contains three 12-bit registers, E0, E1, E2. Register E0 has a preferred status, inasmuch as it feeds the pack logic and is the only exponent register which can be loaded directly into AC or MQ.

At the center of the figure is a 12-bit one's complement adder/subtracter. Like the one in the main arithmetic unit, it is a subtractive adder. Even though this adder (like the one in the main unit) uses carry look-ahead, one clock cycle is not sufficient time to perform an add and gate the results

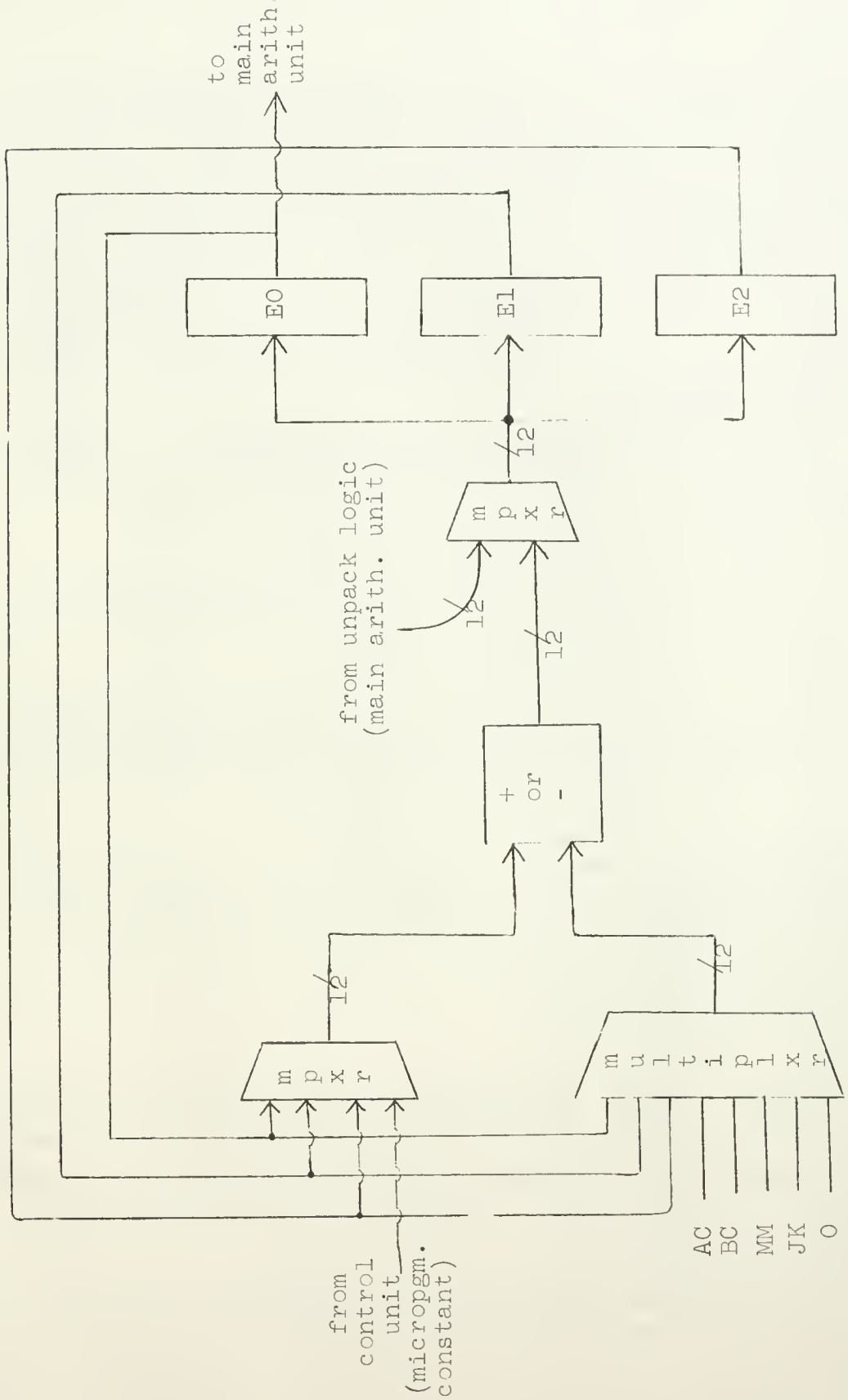

Figure 4. PUMA central processor: exponent arithmetic unit.

into an E register -- two cycles must be allotted. As a result, a fast addition mode is also provided. In hardware terms, the adder consists of three 4-bit ALU chips and the fast mode disables the carry into each chip. In net effect, this means that bits 0-3, 4-7, and 8-11 are added independently by two's complement adders with a carry forced into each low bit (bits 0, 4, and 8). The fast mode is useful in a number of situations, such as when a count smaller than 16 is being decremented.

One of the operands of the adder can be any of the E registers or a constant sent from the control unit. The other operand can be any E register or

the low 12 bits of the AC
a count of the number of 1 bits in the low 4 bits of the AC
the low 4 bits of the MQ in bits 0-3 and the complement thereof in bits 4-7
the jk field (6 bits) of the current instruction
minus zero (all 1 bits)

The output of the adder and the output of the unpack logic in the main arithmetic unit are sent to a multiplexer which transmits one of these two signals to the E register inputs.

4.2 The Instruction Unit

Figure 5 shows the structure of the instruction unit. This unit holds the current instruction and the next instruction in sequence. It also contains the P register (program counter) and MA register, which holds the address for memory references.

The data path from memory is connected to the input of the NIW (next instruction word) register. While one instruction word is executing, the next word is requested from memory. When the word is delivered by the memory, the NIW is clocked to load it. When execution of the current instruction word is

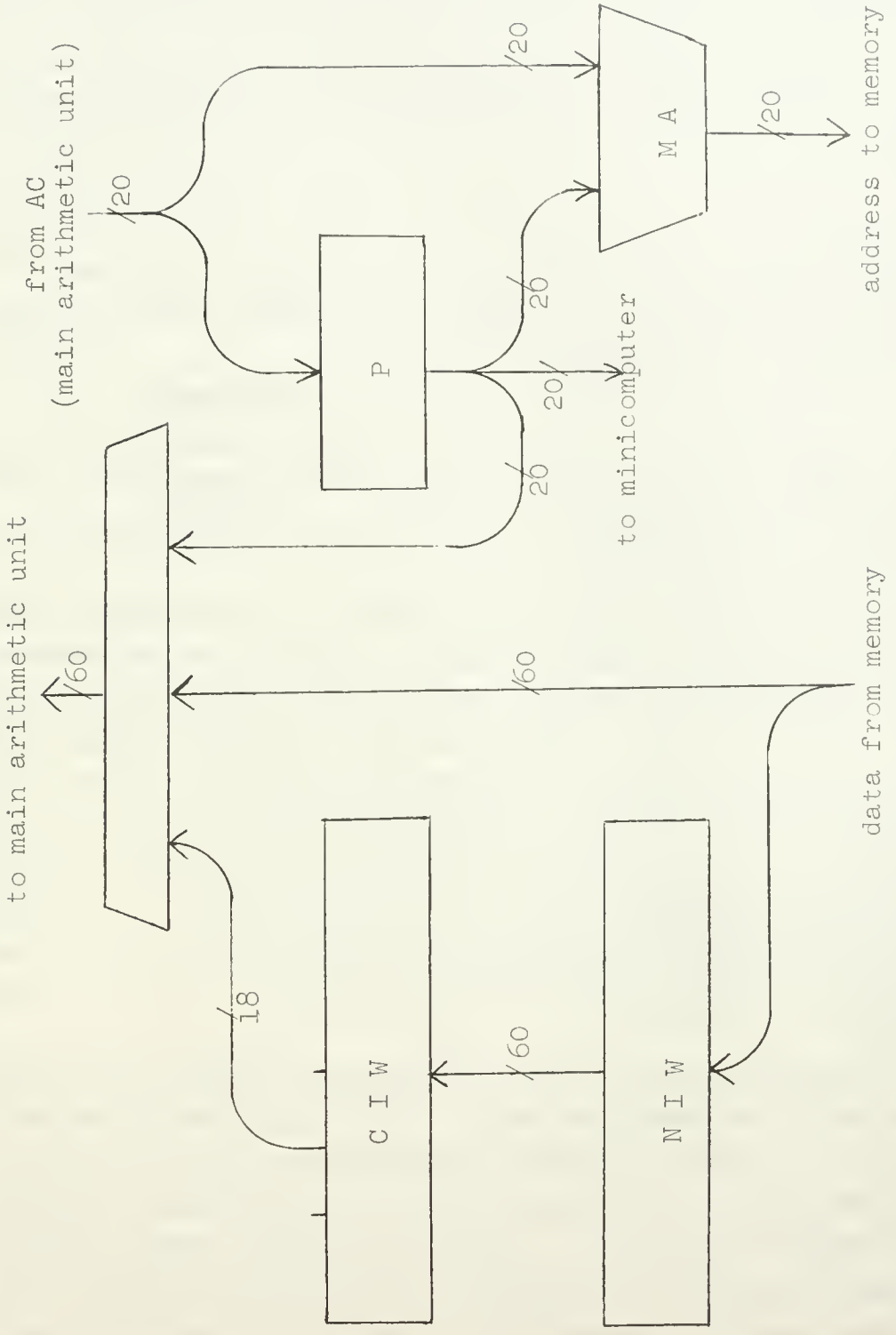

Figure 5. PUMA central processor: instruction unit.

complete, the next instruction word is transferred from the NIW to the CIW (current instruction word) register.

Lines from CIW transmit the high 15 bits (f,m,i,j, and k fields of the first instruction) to the arithmetic and control units. In addition, bits 30 to 47 (the K field of the first instruction) can be routed to the arithmetic unit and or-ed with the low 18 bits of the ALU output. The CIW is wired to perform 15-bit left shifts. After a short instruction (15 bits) is executed, one shift is performed; after a long instruction (30 bits), two shifts.

Controlling the data sent to the arithmetic unit (or-ed with the ALU) is a 4-way multiplexer. One input, as we just mentioned, is the K field of the current instruction. The second is the data sent from memory, the third is the P register, and the fourth is zero.

On the right side of the figure is the address path. Although the 6600 design only has 18 bits for addresses, the PUMA address path has been made 20 bits throughout to accommodate a million words of memory (this involved almost no extra cost, since all the chips involved are 4-bit slices). The P register is a counter — it can be incremented, decremented, or set from the low 20 bits of the AC register. The P register can be directly read by the minicomputer. The MA register holds the address for memory references. It can be loaded from the output of the P register or from the low 20 bits of AC.

For PUMA serial number 2, some additional logic was included in the address path. This logic is controlled from the memory switch and so is invisible at the PUMA microprogram level. Two registers have been added for relocation and memory protection: a reference address (RA) register and a field length (FL) register. The output of the MA register is continuously compared to FL; an address out of range signal is sent to the memory switch if MA $\geq$ FL. The output of MA is added to RA to obtain the absolute memory address, which is sent to the memory switch. Both RA and FL can be set only by

the minicomputer through the memory switch. Finally, to reduce the number of data paths leaving the PUMA, the absolute memory address, P, RA, and FL registers are multiplexed onto a single 20-bit data path under control of the memory switch.

4.3 The Control Unit

The control unit generates the signals which control the arithmetic and instruction units. It must generate these signals in the proper sequence to interpret and emulate CDC 6600 instructions.

The control unit of the PUMA is microprogrammed — all information about the sequencing of control signals is stored in a *microprogram memory*. The microprogram consists of a series of *microinstructions*. Each microinstruction specifies the values of the control signals during a single clock cycle and contains sequencing information which determines the microinstruction to be used in the following cycle. This sequencing information consists of

a condition number, which selects one of 48 conditions to test
the address of the microinstruction to be executed next if the condition is true
the address of the microinstruction to be executed next if the condition is false

Including a conditional branch in every microinstruction requires a much wider microinstruction than would having separate operation and branch microinstructions. However, because of the highly branched nature of the microprogram, including the conditional branch in each microinstruction makes the microprogram much shorter and hence much faster. Since speed is paramount in this design, this approach has been selected.

We have designed two versions of the control unit. One version incorporates a 1024-word *writeable* microprogram memory. This version has been used to debug the microcode for 6600 emulation, and will probably be used in the future to test experimental microprograms. This version has the disadvantage of being relatively slow — the clock period is 250 ns. In the second version of the control unit, the microprogram is stored in a *programmable read-only* memory. This memory is smaller (512 words) and, of course, cannot be changed once programmed, but makes for a much faster control unit — a clock period of about 50 ns. We shall refer to these two versions as the slow and fast control units, respectively.

The microinstructions are of a horizontal or decoded form. That is, there is (with a few exceptions) a one-to-one correspondence between bits in the microinstruction and signals coming out of the control unit. We may also note here that the entire processor uses a single (one-phase) clock, and that control signals remain constant during a clock period.

We shall now proceed to a detailed description of the control units — first the slow version, and then the fast.

The gross structure of the slow control unit is shown in Figure 6. At this heart is a microprogram memory of 1024 85-bit words (the memory is built from 1024-bit static RAM chips with a 70 ns maximum access time). The memory and the microinstruction sequencing logic are all TTL rather than ECL, because of the low cost and ready availability of the TTL RAMs (low-cost ECL 1024-bit RAMs were just becoming available when the slow unit was built).

Each 85-bit microinstruction has two parts: a 28-bit group (consisting of a condition field and two branch addresses) which determines the next microinstruction address, and a 57-bit group which determines the value of all control lines during a cycle. These control lines go to the arithmetic and instruction units and a few circuits on the control unit, to be described later. Since the other units require ECL-level control lines, the output of these bits of the microprogram

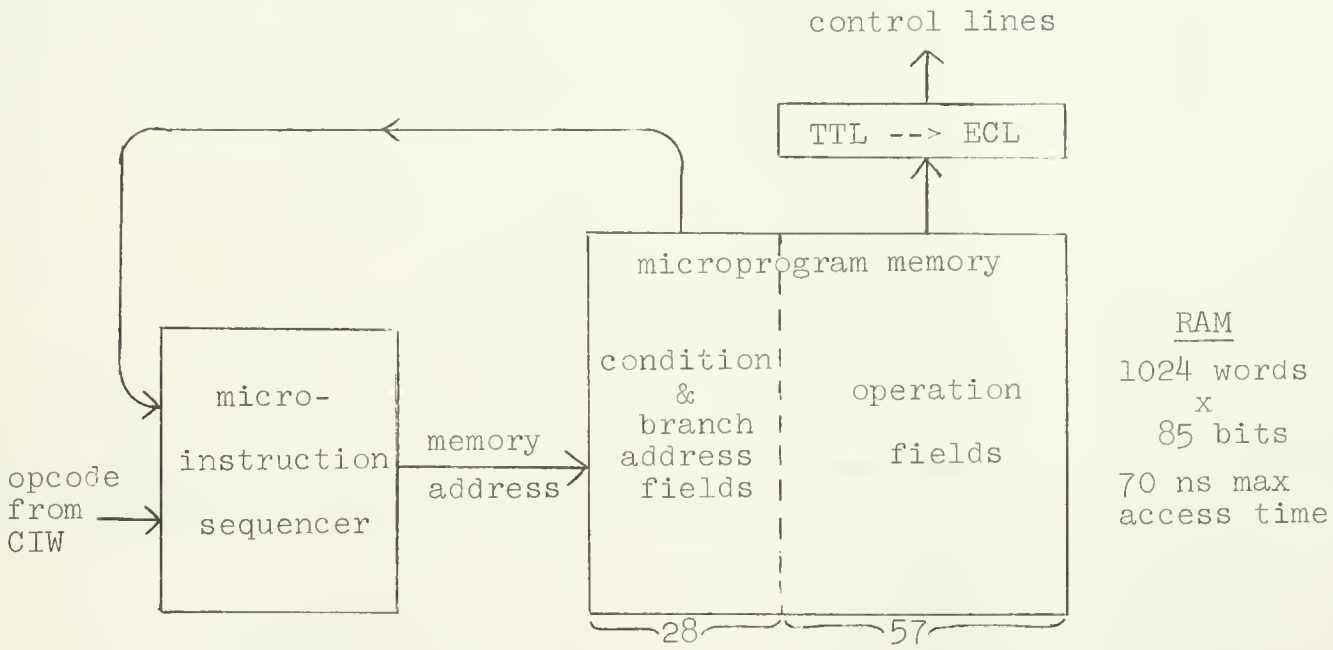

Figure 6. Gross structure of slow control unit.

memory must go through TTL to ECL level converters.

Writing of the microprogram memory is under the control of the minicomputer. The minicomputer has three 12-bit paths to the control unit: a path for data into the control unit, a path for data out of the control unit, and a path for function signals (to start the clock or write a micro-instruction). The minicomputer can write a word of the micro-program memory by sending eight 12-bit data words, together with the appropriate function signals.

Figure 7 shows the details of the next microinstruction sequencing logic. As is our usual habit, we shall work our way through this diagram from left to right. Coming in on the left is the 28-bit group from the current microinstruction. This group is composed of three fields:

a 6-bit condition field

an 11-bit true-branch-address field

an 11-bit false-branch-address field

The high-order bit of each branch address has a special function: if it is set, the low 6 bits of that branch address are to be replaced by the opcode of the instruction currently being interpreted (the high 6 bits of register CIW). This feature permits a rapid transfer to the appropriate microcode sequence for each new instruction.

The condition field selects the signal to be tested by the current microinstruction. There are 47 conditions in PUMA: 31 test lines on the arithmetic unit; 16 test signals generated on the control unit. These 47 signals are fed into a giant multiplexer circuit (actually distributed between the arithmetic and control units), whose output is the signal to be tested. This signal in turn feeds a multiplexer which selects between the true-branch-address and the false-branch-address. The output of this multiplexer is the address of the next microinstruction to be executed. This address is gated into the microinstruction address register (MAR) and is also available to be read by the minicomputer.

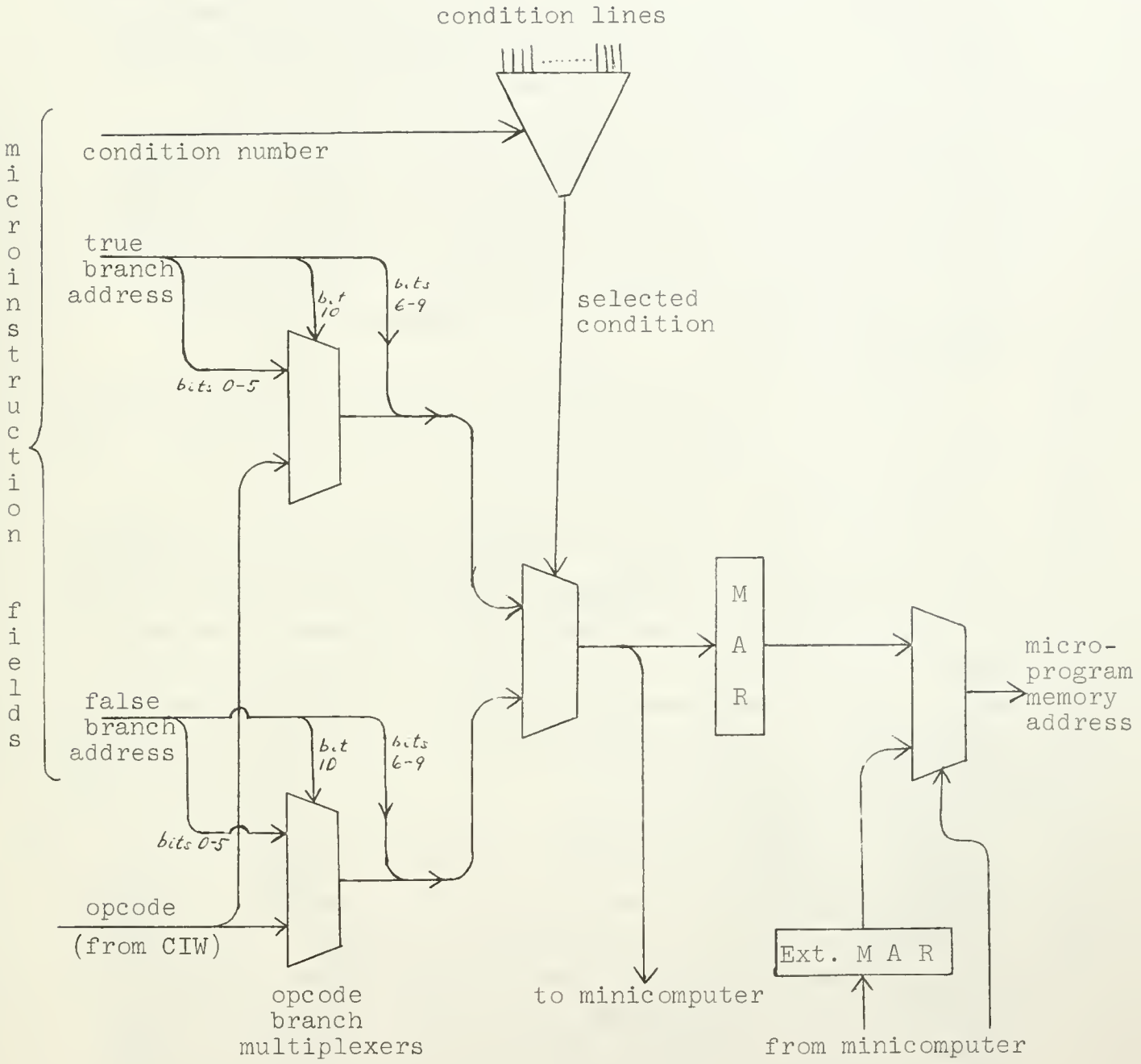

Figure 7. Microinstruction sequencer.

A final multiplexer selects between the microinstruction address register and the external microstore address register (Ext. MAR). The external address register can be set by the minicomputer, and the select line of this multiplexer is under minicomputer control. The external address register is selected when the microprogram memory is being written, and for the first cycle when the processor is being started. When the processor is running, the microinstruction address register is selected. The output of this multiplexer finally feeds the address lines of the microprogram memory.

There are a few exceptions to the rule that each control line is directly controlled by one microinstruction bit. These exceptions are detailed in Figure 8.

One exception is the set of "special functions". These are control signals which are changed relatively infrequently in the microprogram. They are: the lines controlling the NIW, P, and MA registers; the lines controlling memory requests; and the line to the "save PG" flip-flops in the main arithmetic unit. To save some space in the microinstruction word, the bits of the microinstruction which are used as the constant in the exponent arithmetic unit also control the special function lines. The circuit works as follows: the microinstruction includes a special function bit. If this bit is 0, the special function lines are all forced to 1 (which causes all the registers to hold their previous contents). If this bit is 1, the special function lines are controlled by bits of the constant field. Thus the only constraint imposed by this scheme is that the constant field cannot be used in the same cycle as a special function (unless the constant required just happens to be that produced by the pattern of special functions).

The other irregularity concerns the lines which control the register number in the main arithmetic unit. The microinstruction can specify this number by a constant in the

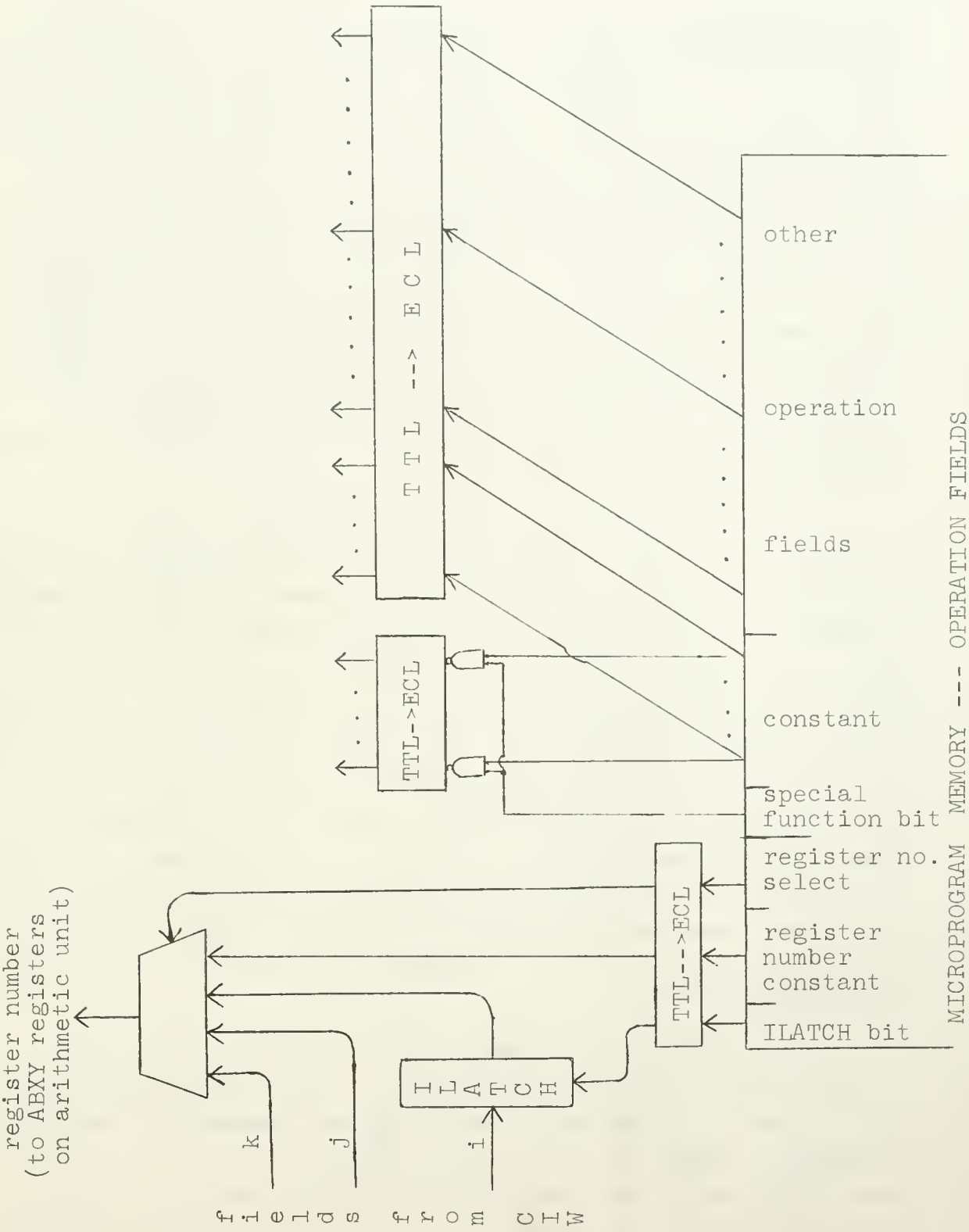

Figure 8. Details of microprogram memory output logic.

microinstruction, or it can specify that the i, j , or k field of the current instruction (bits 51-53, 48-50, or 45-47 of CIW) be used as the register number. Accordingly, there is a 3-bit, 4-to-1 multiplexer feeding the register number. Note also the 3-bit register ILATCH, which is loaded with the i field of the current instruction (bits 51-53 of CIW). ILATCH makes it possible to retain the i field of the current instruction while rotating CIW in preparation for executing the next instruction. (The j and k fields are not similarly latched because the i field is frequently used at the end of the microcode sequences for instruction interpretation, whereas the j and k fields are most often used at the beginning of such sequences.)

The fast control unit has been designed to minimize the machine cycle time. One way in which this was done was to use a very fast chip in the microprogram memory. The chip we have chosen is a 256 word × 4 bit PROM (programmable read-only memory) with a maximum access time of 25 ns. Another way in which we speeded up the control unit was to overlap operations in the arithmetic unit with the fetch of the next microinstruction. This is a bit tricky, since the next microinstruction selected may be determined by a condition which depends on the result of an arithmetic operation. What we were forced to do was fetch *both* possible successor microinstructions and then, near the end of the clock cycle (when the value of the condition was known) select one of the two as the actual successor.

The circuit which achieves this overlap in the fast control unit is shown in Figure 9. The microprogram memory is divided into two banks which can be accessed concurrently. We require that the false-branch-address always be in bank 0 and the true-branch-address always be in bank 1. In this way the two possible successors of a microinstruction can always be fetched concurrently. (This means that microinstructions which occur as the false-branch-address of one microinstruction and the true-branch-address of another microinstruction

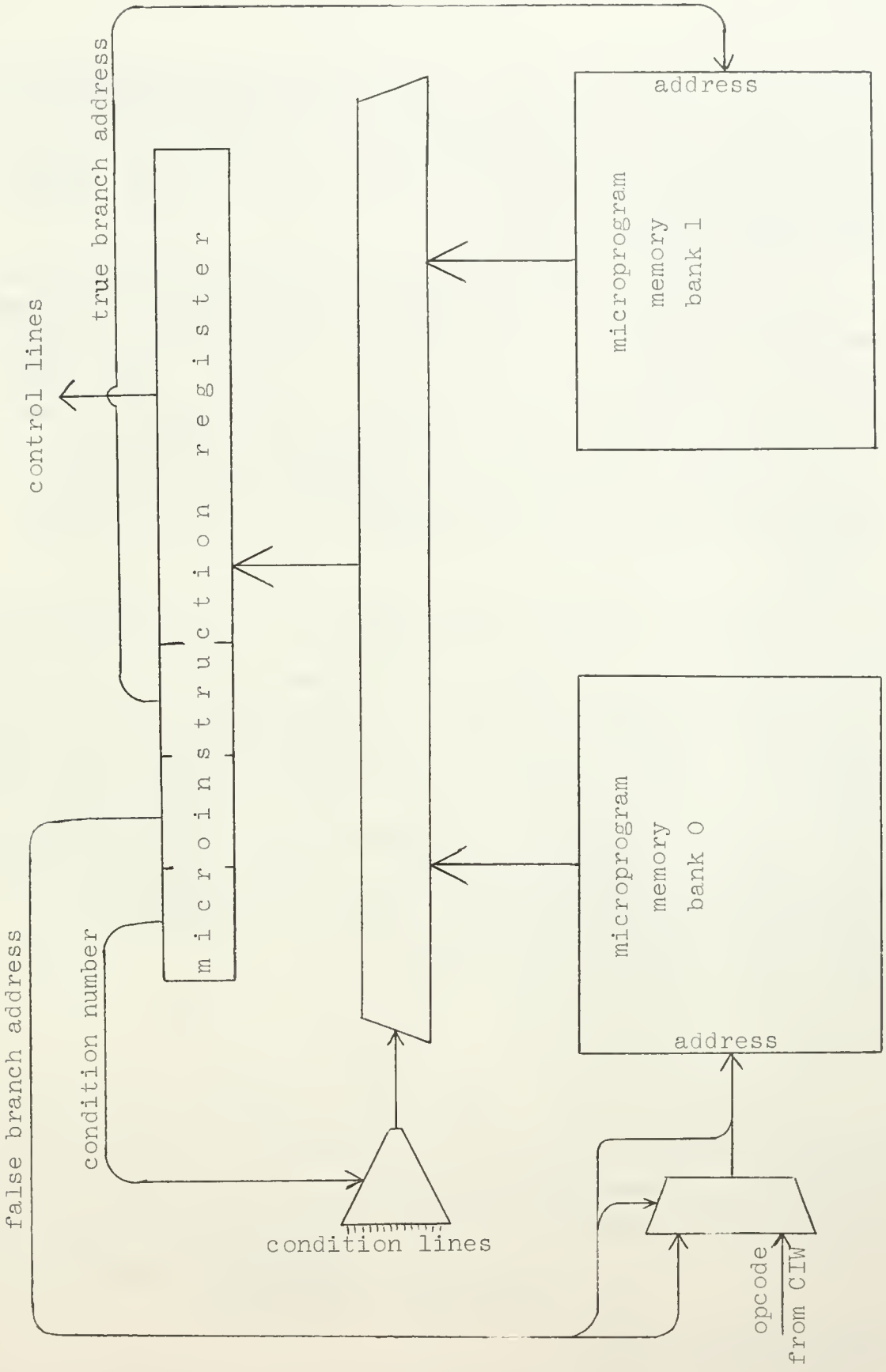

Figure 9. Fast control unit.

will have to appear twice in microprogram memory, once in each bank. In practice, however, the number of microinstructions which must be thus duplicated is quite small.) The multiplexer which permits branching on the current opcode is included only in bank 0.

During the first 35 ns or so of a clock cycle, the arithmetic operation is performed and both potential successor microinstructions are fetched. The specified condition is then selected by the condition multiplexer. The output of this multiplexer is used as the select input of a 2-way, 81-bit multiplexer to select the next microinstruction, which is then gated into the microinstruction register. This takes about 15 ns, for a total cycle time of about 50 ns.

In contrast to the slow control unit, this design does not provide a multiplexer for introducing an external microprogram address. Instead, the minicomputer can disable the microprogram memory and write the microinstruction register directly. Thus, to start the machine the minicomputer would load the first microinstruction into the microinstruction register and then start the clock running. This facility for writing the microinstruction register makes it possible to try microinstructions not in the microprogram memory; this may be useful for diagnostic purposes.

5. Microprogramming the PUMA

For convenience in microprogramming the PUMA, we have provided a microassembly language similar in style to a register transfer language or algebraic programming language. In this chapter we shall describe the PUMA micro-operation repertoire in terms of this microassembly language. An appendix to this chapter describes the structure of the assembler microinstructions, explaining the significance of each bit.

The microassembly language is processed by a micro-assembler. The microassembler was originally written in SNOBOL 4; the current version was coded in CIMS PL/I by Erdwin Chua. The output of the microassembler may be fed to one of two postprocessors. One postprocessor, for the slow control unit, has the task of permuting the microinstructions so that the first microinstruction in the sequence which interprets opcode n ($n=0,\ldots,77_8$) is placed at location n of the microprogram memory. The other postprocessor, for the fast control unit, has the *additional* task of assigning microinstructions to one or both banks of the control store, based on an analysis of branching in the microprogram.

To summarize the data register complement of the PUMA central processor: the PUMA provides 16 60-bit registers numbered X0-7, Y0-7; 16 20-bit registers numbered A0-7, B0-7; a 60-bit transfer register BUF; a 120-bit AC:MQ register pair consisting of the two separately usable halves AC and MQ, and 3 12-bit registers used for exponent arithmetic and miscellaneous counting operations, which are numbered E0, E1, and E2. Within an n-bit register, the least-significant bit is labeled bit 0 and the most-significant bit is labeled bit n-1. The micro-operations of the PUMA are as follows:

(a) Write to register:

{A|B|X|Y} {r|I|J|K|M|N} = AC

or

= E0:AC (pack option)

Here and below, r is a digit between 0 and 7; I, J, and K

designate the 3-bit I, J, and K fields of the current instruction; whereas M and N use bits from the E1 register, with some encoding, in the following special way (designed for the multiply routines):

M: if bit 7 of E1 is on, select register using bits 0-2 of E1, otherwise no register store takes place

N: if bit 3 of E1 is on, select register using bits 4-6 of E1, otherwise no register store tackes place.

For a pack option write to register, bits 0-47 and 59 of E0:AC are set from AC, and bits 48-58 from E0, with bit 58 being complemented. In addition, if bit 59 of AC is on, the exponent bits 48-58 are complemented before the write. This option stores in the register the 6600 floating-point representation of the number whose integer coefficient is in AC and exponent is in E0.

(b) Read from register:

These operations have BUF as their target; register read and write operations cannot both be performed on the same cycle. Microcode forms are

BUF = {A|B|X|Y} {r|I|J|K|M|N}

or

Ei:BUF = {A|B|X|Y} {r|I|J|K|M|N} (unpack option) i=0,1 or 2

Here r,I,J,K,M,N are as above; in the M (resp. N) case 0 will be loaded into BUF if bit 7 (resp. bit 3) of E1 is off. In the case of a read from an A or B register, bits 20-59 of BUF will be set to 0.

On the unpack option, the sign of the quantity being read is extended to fill bits 48-58 of BUF, while Ei is set as follows: Ei(0-9) = source(48-57)/source(59); Ei(10)=Ei(11) = *not*(source(58)/source(59)), where "/" indicates exclusive-or. This option separates a number in 6600 floating-point representation into its integer coefficient (BUF) and exponent (Ei).

(c) Set AC and MQ:

A 60-bit arithmetic-logical unit can compute any of several functions of AC and BUF. The output of the arithmetic-logical unit, together with the output of the MQ register,

is fed to a 120-bit-wide shift unit capable of shifting 60, 16, 4 or 0 bit positions. The two halves of the AC:MQ register are separately setable from the output of the shift unit.

Several inputs besides the functions of AC and BUF can be fed to the shift unit. In addition, AC:MQ can act by itself as a shift register, shifting one bit at a time in either direction. Any of these operations can be performed in parallel with a BUF load or register-write operation. Logical operations can be performed in one cycle; arithmetic operations, however, require two cycles.

The microcode forms for these operations are:

```
{AC|MQ|AC:MQ|(AC)} = f
or
                   = SHIFT(f:MQ, longsh)
or
                   = MQ
or
                   = SHIFT(AC:MQ, shortsh)
```

The left side of the assignment indicates which of AC and MQ will be loaded on this cycle; "(AC)" indicates that a function should be computed by the arithmetic-logical unit, but not loaded into either register half on this cycle. This is necessary for arithmetic functions, which take two cycles to compute. For example, to add BUF to AC the two-cycle microinstruction sequence

```
(AC) = AC + BUF
 AC  = AC + BUF
```

is required; the sum is not loaded into the AC until the end of the second cycle.

"f" may be any Boolean function of AC and BUF; mnemonics are currently provided in the microassembler for the following:

```
AC
BUF
AC ∧ BUF
AC ∨ BUF
AC / BUF        (exclusive or)
AC ∧ ¬ BUF
¬ AC
¬ BUF
0
- 0            (all ones)
```

"f" may also be one of the following arithmetic operations.

```
AC + BUF
AC - BUF
AC + 0
AC - 0
```

Addition is normally done in 60-bit one's complement arithmetic using a subtractive adder; that is, the sum is effectively computed by complementing both operands, performing an addition with end-around carry, and complementing the result. This scheme, which is also used on the 6600, yields +0 when adding a number and its complement. Several arithmetic options are available and are indicated by appending one or more of the following microcode suffices to the operation.

[18]	perform 18-bit 1's complement arithmetic, with the sign of the result extended to the high-order 42 bits
[NOP]	suppress end-around carry ("no propagate")
[G]	force carry into low bit ("generate")
[SAVEPG]	save 60-bit carry generate and propagate bits (does not affect result of current operation)
[USEPG]	make end-around carry conditional on values of P and G saved by SAVEPG: carry into low bit = (carry out of high bit ∧ P)∨G (SAVEPG and USEPG are provided to perform 120-bit one's complement addition efficiently.)

Finally, "f" can be one of the following.

K	18-bit K field of instruction in bits 0-17, with bits 18-59 zero
P	contents of P register (address of current instruction word + 1) in bits 0-19 with bits 20-59 zero
E0	contents of E0 in bits 0-11, with sign bit extended into bits 12-17 and zeroes in bits 18-59
CMRD	word of data read from central memory

"longsh" may indicate one of the following four shifts to be performed by the shift unit.

R4	right circular shift 4 bits
R16	right circular shift 16 bits
A4	arithmetic right shift 4 bits (the sign bit, AC bit 59, is extended to fill vacated bit positions)
A16	arithmetic right shift 16 bits

The shift unit can also perform a 60-bit circular shift, directing the output of the arithmetic-logical unit to the MQ and the output of the MQ to the AC. This shift is selected by the microinstructions

```
AC = MQ
```

and

```
MQ =  f
```

which may be executed simultaneously.

AC:MQ can also act as a 120-bit shift register, performing a one-bit shift on each cycle. In one-bit shifts, data does not pass through the arithmetic-logical and shift units. "shortsh" may select one of the following one-bit shifts.

R1	right circular shift 1 bit
A1	right arithmetic shift 1 bit
L1	left circular shift 1 bit
Z1	left shift 1 bit, zero fill
O1	left shift 1 bit, one fill

(d) Exponent unit operations:

The exponent unit performs addition and subtraction of 12-bit quantities using a one's complement subtractive adder. Exponent unit operations may proceed in parallel with register and AC:MQ operations, with one exception: because only one exponent register may be loaded in any cycle, a BUF load with unpack option excludes a simultaneous exponent unit assignment.

The form of an exponent unit operation is

$$[Ei] = \{Ej|const\} \pm \{Ek|AC|JK|-0|MM|BC\}$$

or

$$[Ei] = \{E_j|const|AC|JK|MM|BC\}, \quad i,j,k=0,1 \text{ or } 2$$

Here,

const	is a 12-bit octal constant (the microassembler interprets -const as the one's complement)
AC	bits 0-11 of AC
JK	6-bit jk field of current instruction in bits 0-5; zeros in bits 6-11
-0	all one bits
MM	in bits 0-3, MQ bits 0-3; in bits 4-7, complement of MQ bits 0-3; in bits 8-11, zeros
BC	bit count of low 4 bits of AC

(The operand -0 is not explicitly available to the microassembly language programmer. However, the single-operand assignments

$$E_i = \{E_j \mid const\}$$

are assembled as

$$E_i = \{E_j \mid const\} + (-0)$$

since $-0 = 7777_8$ is the additive identity. Similarly,

$$E_i = \{AC|JK|MM|BC\}$$

is assembled as

$$E_i = 7777 + \{AC|JK|MM|BC\} .)$$

As in the case of the main arithmetic-logical unit, addition and subtraction require two cycles. An empty left-hand side

indicates that the arithmetic operation is to begin but no result is to be stored on this cycle; thus a two-cycle add might be

$$= AC + 1$$
$$E0 = AC + 1$$

A "fast add" option, indicated by [F] after the operation, is also provided; a fast add can be performed in one cycle. In a fast add, the carry into bit position 0, 4, and 8 is blocked (set to 0), so the 12-bit adder acts like three separate 4-bit adders.

(e) Tests and transfers in the microcode:

A test can be used during any PUMA microinstruction cycle to select one of two microcode successor addresses for a given instruction. The general test form is IF t THEN L1 ELSE L2, where L1 and L2 are the labels fof two microinstructions.

The following 48 tests are provided:

NULL	false
EALU(11)	EALU highbit on
EALU(0) ∨ EALU(1)	EALU lowbits not zero
EALU(2) ∨ EALU(3)	EALU bits 2-3 not both zero
EALU(4) ∨ EALU(5)	EALU bits 4-5 not both zero
EALU(0-3)	EALU bits 0-3 not all zero
EALUPOUT	carry propagate bit for EALU on
EALUPOUT ∧ EXOP2.10	EALU carry propagate and EALU 2nd op, bit 10
EALUPOUT ∧ ¬EXOP2.10	EALU carry propagate and not EALU 2nd op bit 10
FOFL	floating overflow: EALU output outside valid range of $\pm 2000_8$
XFOFL	extreme floating overflow; EALU output outside range $\pm 3000_8$
REG(59)	sign bit on in selected register
REG(17)	bit 17 on in selected register
BUF(59)	sign bit on in BUF

REG(59)/BUF(59)	sign bit of selected register ≠ sign bit of BUF
AC(59)	AC sign bit on
AC(49)	AC bit 49 on
AC(47)	AC bit 47 on
AC(46)	AC bit 46 on
MQ(59)	MQ sign on
MQ(50)	MQ bit 50 on
MQ(49)	MQ bit 49 on
ALU(59)	ALU sign bit on
ALU(49)	ALU bit 49 on
ALU(47)	ALU bit 47 on
ALU(59)/ALU(47)	ALU sign different from ALU bit 47
ALU(59)/ALU(48)	ALU sign different from ALU bit 48
POUT	propagate output on in ALU
G	generate output on in ALU
M	low 4 bits of MQ exceed 7 (used in multiply routine)
M > 8	low 4 bits of MQ exceed 8 (used in multiply routine)
AC << BUF ∧ ¬MQ(50)	weird condition used in divide routine, see divide documentation
OPCODE(0)	low bit of opcode on
OPCODE(1)	bit 1 of opcode on
OPCODE(2)	bit 2 of opcode on
I(0)	low bit of I field on
I(1)	bit 1 of I field on
I(2)	bit 2 of I field on
I = 0	I field 0
I > 5	I field exceeds 5
J = 0	J field 0
MODE2	external line (set by PP)
MODE4	external line (set by PP)
CMDONE	memory operation complete
NIWEMPTY	next instruction word buffer empty (see below)
LASTPARCEL	last parcel of instruction word
ICHECK	LASTPARCEL ∨ (NIWEMPTY ∧ CMDONE)
EXTINT	external line (set by PP)

The propagate outputs of the arithmetic-logical unit and the exponent unit adder are very useful in performing selective bit tests. The propagate output when adding A and B is

$$(\bar{A}_0 \vee \bar{B}_0) \wedge (\bar{A}_1 \vee \bar{B}_1) \wedge \ldots \wedge (\bar{A}_n \vee \bar{B}_n)$$

(the A and B bits are complemented because this is a subtractive adder), while for subtracting A - B it is

$$(\bar{A}_0 \vee B_0) \wedge (\bar{A}_1 \vee B_1) \wedge \ldots \wedge (\bar{A}_n \vee B_n)$$

Thus, to test that certain bits in B are all 1, we set the corresponding bits in A to 1, compute A - B and test the propagate output.

Similarly, to test that certain bits in B are all 0, we set the corresponding bits in A to 1, compute A + B and test the propagate output. A number of conditions which are tested in this way and used in the microcode have been assigned separate mnemonics; these include

AC = 0	all bits of AC = 0
INDEF(Ei)	Ei has the value 7777_8 (= -0, the "indefinite" exponent)
INF(Ei)	Ei has the value 1777_8 or 5777_8 (the exponent for floating-point infinity)
ILL(Ei)	INF(Ei) ∨ INDEF(Ei)
ZERO(Ei)	Ei has the value 2000_8 or 6000_8 (the exponent for floating-point zero)
E0(6-10) = 0	bits 6 through 10 of E0 are off
E2(7-11) = 0	bits 7 through 11 of E2 are off
E2(6-11) = 0	bits 6 through 11 of E2 are off
E2 = 0	all bits of E2 are off

Note that the first of these conditions requires the use of the arithmetic-logical unit and the remainder the use of the exponent unit.

In addition to the basic form

IF t THEN L1 ELSE L2

the assembler allows the form

IF ¬t THEN L3 ELSE L4

which is assembled as

IF t THEN L4 ELSE L3,

and the form

IF t THEN L5

which is assembled as

IF t THEN L5 ELSE next instruction.

Finally, the form

GO L6

is allowed to indicate unconditional transfer.

A microinstruction transfer to the special label OPCODEBRANCH is detected by the hardware and causes a branch to microinstruction $0\text{-}77_8$, as determined by the opcode field of the current instruction.

(f) Central memory communication:

A central memory operation is initiated by passing the relevant memory address from AC into the memory address register MA, and then setting one of the memory control flip-flops READ/FF, WRITE/FF to signal the operation desired, for which the two micro operations READ and WRITE are provided. On a WRITE operation, the data written is taken from the AC, which should be set from BUF no later than the cycle on which the WRITE operation is executed; AC must then remain unchanged until the memory unit signals completion of the write operation. The normal initiation of write and read respectively is therefore

(write) MA = AC; AC = BUF; WRITE

(read) MA = AC; READ

The memory unit signals the availability of a read result or the acceptance of write-data by raising a CMDONE flipflop. In the case of a read operation, the microprogram must load the data from memory by executing AC = CMRD. The microprogram then acknowledges to the memory its receipt of the CMDONE signal by clearing the READ and WRITE FFs with the CLEAR micro-operation. The memory concludes the cycle and signals its availability by dropping the CMDONE signal.

The basic microprogram sequence for a read, beginning with the address in AC and ending with the data in AC, is

```
LOWAIT1   IF CMDONE THEN LOWAIT1
          MA = AC;  READ
LOWAIT2   IF ¬CMDONE THEN LOWAIT2
          AC = CMRD;  CLEAR
```

For a write the sequence, beginning with the address in AC and the data in BUF, is

```
STRWAIT1  IF CMDONE THEN STRWAIT1
          MA = AC;  AC = BUF;  WRITE
STRWAIT2  IF ¬CMDONE THEN STRWAIT2
          CLEAR
```

Address relocation and address-out-of-range detection is the responsibility of the memory unit, not the microprogram.

(g) Instruction fetch logic:

In PUMA, instructions are read out of a 60 bit CIW (current instruction word) register which is addressed in 15 bit parcels by a parcel counter. A 60 bit backup register NIW ("next instruction word") is also provided. A 20 bit P register contains the address of the current instruction + 1. Operations available at the microprogram level are

```
NEWPARCEL    (shift CIW left 15, decrement parcel counter
             by 1)
CIW = NIW    (move instruction backup word to CIW;
             set NIW empty and parcel counter = 4)
NIW = CMRD   (fill instruction backup word, set NIW full)
```

```
P = P + 1      (increment instruction counter)
P = AC         (set instruction counter from AC, i.e. branch)
MA = P         (set memory address register from P)
LATCH I        (load I-field latch)
```

The opcode and J and K fields of the current instruction are read directly from the CIW register. References to the I-field, however (in register reads and writes and in conditions), actually use the contents of the I latch. The I latch is loaded from the I field of the current instruction by the LATCH I micro-operation. This latch was included in PUMA because it was desirable in the microcode to reference the I field of the current instruction after the CIW had been rotated to the next instruction by a NEWPARCEL operation.

Testable conditions involving the instruction unit are

LASTPARCEL	parcel counter = 0
NIWEMPTY	NIW empty (CIW = NIW was done with no concurrent or subsequent NIW = CMRD)
ICHECK	LASTPARCEL ∨ (NIWEMPTY ∧ CMDONE)

The utility of condition ICHECK is discussed in the next chapter.

Microassembler input format

A microinstruction is a set of one or more micro-operations to be performed in a single clock cycle. The form of a microinstruction is

```
[label]  micro-operation  [; micro-operation] ...
```

A microinstruction must be punched starting on a new card. If the instruction has a label, the label must begin in column 1 and must be separated from the first micro-operation by one or more blanks. If there is no label, column 1 must be blank, and the first micro-operation may start in or after column 2.

Except for these restrictions, input is free-format within columns 1 to 80 and blanks may be freely used to improve readability; blanks may not appear, however, within labels, keywords, or register names. An instruction which does not fit on one card may be continued by placing a '+' in column 1 of the following card and continuing the instruction in or after column 2. Anything following an * on a card is treated as a comment.

The two-digit labels 00 through 77_8 have a special significance: the instruction labeled ij should be the first microinstruction in the sequence for processing the machine language instruction with opcode ij; this microinstruction will be placed in location ij of the microprogram memory.

APPENDIX

P U M A M I C R O I N S T R U C T I O N F I E L D S

THE FIELD NAMES LISTED ARE THE NAMES OF THE CORRESPONDING CONTROL LINE NAMES IN PUMA SER. NO. 1. THE BIT POSITIONS GIVEN ARE THOSE FOR THE SLOW CONTROL UNIT; THOSE FOR THE FAST CONTROL UNIT DIFFER IN HAVING 8-BIT BRANCH ADDRESSES (FIELDS BRADDR0 AND BRADDR1) AND NO OPCDBR1 FIELD.

FIELD	(WIDTH IN BITS)	SIGNIFICANCE	BIT POSITION
MAIN AU			
REGISTERS			
READTP	(1)	0→WRITE REGISTER, 1→READ REGISTER	37
XYS-TP	(1)	0→SELECT XY REGISTERS, 1→SELECT AB REGISTERS	36
YBSTP	(1)	0→SELECT AX REGISTERS, 1→SELECT BY REGISTERS	35
REGCNST	(3)	SPECIFIES CONSTANT REGISTER NUMBER	22
IJKSEL	(2)	SELECTS (3-BIT) FIELD WHICH DETERMINES REG. NO. 0→USE K FIELD OF CURRENT INSTRUCTION 1→USE J FIELD OF CURRENT INSTRUCTION 2→USE CONTENTS OF I LATCH 3→USE REGCNST FIELD OF MICROINSTRUCTION	25
MORNSTP	(1)	1→USE M OR N AS REGISTER NUMBER	65
NREGSTP	(1)	0→SELECT M AS REG. NO., 1→SELECT N AS REG. NO. NOTE: IF M OR N IS TO BE USED AS REGISTER NO., IJKSEL SHOULD =3 AND REGCNST=0	66
BUF			
EXLSTP	(1)	1→SELECT EXPONENT EXTRACT (OUT OF REGISTER) AND EXPONENT MERGE (INTO REGISTER)	56
LDBUFTP	(1)	0→LOAD BUFFER, 1→HOLD	64
ALU			
ALUSTP	(5)	SELECT ALU FUNCTION; ALU FUNCTIONS ARE FUNCTIONS OF THE 10181 ALU CHIP, AS FOLLOWS: 0 → AC+0 6 → AC+BUF 9 → AC-BUF 15 → AC-0 16 → -AC 19 → -0 21 → -BUF 25 → AC/BUF [EXCLUSIVE-OR] 26 → BUF 27 → AC∨BUF 28 → 0 29 → AC∧-BUF 30 → AC∧BUF 31 → AC NOTE: OUTPUT OF ALU IS OR-ED WITH OUTPUT OF E0 TO AC GATES AND DATAIN MULTIPLEXOR ON IU, SO ZERO FUNCTION OF ALU (28) MUST BE SELECTED WHEN OUTPUT FROM ONE OF THESE OTHER SOURCES IS USED	42
ALU16TP	(1)	0→60 BIT ARITHMETIC; 1→16 BIT ARITHMETIC	34
PGSTP	(2)	CONTROLS END-AROUND CARRY PROPAGATION: 0 → NO END-AROUND CARRY PROPAGATION [NOP]	62

```
                              1 ↱ PROPAGATE CARRY END-AROUND
                              2 ↱ GENERATE CARRY INTO LOW BIT [G]
                              3 ↱ USE SAVED P AND G [USEPG]
    *SAVETP-   (1)      0↱HOLD LAST SAVED GENERATE AND PROPAGATE VALUES    77
                        1↱SAVE NEW GENERATE AND PROPAGATE VALUES FROM ALU
    ETOACTP    (1)      1↱GATE E0 ONTO LOW 12 BITS OF ALU OUTPUT BUS      61
  SHIFTER
    SHSTP      (2)      DETERMINES AMOUNT OF SHIFT:                       51
                              0 ↱ NO SHIFT
                              1 ↱ SHIFT 4 BITS RIGHT
                              2 ↱ SHIFT 16 BITS RIGHT
                              3 ↱ SHIFT 60 BITS
    LSHSTP     (2)      FOR RIGHT SHIFTS, SELECTS TYPE OF SHIFT:          67
                              0, 1 OR 2 ↱ RIGHT CIRCULAR SHIFT [R]
                              3 ↱ RIGHT ARITHMETIC SHIFT [A]
                        FOR 1-BIT LEFT SHIFTS, DONE IN AC:MQ, DETERMINES
                        BIT TO FILL LOW BIT OF MQ:
                              0 ↱ FILL WITH 1 BIT [O]
                              1 OR 3 ↱ FILL WITH 0 BIT [Z]
                              2 ↱ FILL WITH VALUE OF AC.59 [L]
  AC+MQ
    ACFTP      (2)      AC FUNCTION:                                      47
                              0 ↱ LOAD
                              1 ↱ LEFT SHIFT 1 BIT
                              2 ↱ RIGHT SHIFT 1 BIT
                              3 ↱ HOLD
    MQFTP      (2)      MQ FUNCTION (SAME CODES AS ACFTP)                 49
EXPONENT AU
  EAU ADDER
    OP1STP     (2)      SELECTS FIRST OPERAND OF EXPONENT ADDER           57
                              0 ↱ CONSTANT FIELD OF MICROINSTRUCTION
                              1 ↱ E0
                              2 ↱ E1
                              3 ↱ E2
    OP2STP     (3)      SELECTS SECOND OPERAND OF EXPONENT ADDER          53
                              0 ↱ ALL ONES
                              1 ↱ E0
                              2 ↱ E1
                              3 ↱ E2
                              4 ↱ LOW 12 BITS OF AC
                              5 ↱ JK FIELD OF CURRENT INSTRUCTION
                              6 ↱ BIT COUNT OF LOW 4 BITS OF AC
                              7 ↱ MM
    ESUBTP     (1)      EXPONENT ADDER FUNCTION:  0↱ADD, 1↱SUBTRACT       59
    EFASTTP    (1)      0↱NORMAL ADD, 1↱ FAST ADD: SUPPRESS CARRIES INTO  60
                        BITS 0, 4, AND 8 OF ADDER
  EXPONENT REGISTERS
    E0CLKTP    (1)      0↱STORE IN E0;  1↱DON≠T STORE IN E0               82
    E1CLKTP    (1)      0↱STORE IN E1;  1↱DON≠T STORE IN E1               83
    E2CLKTP    (1)      0↱STORE IN E2;  1↱DON≠T STORE IN E2               84
  CONSTANT
    CONTP      (12)     12-BIT CONSTANT FOR EXPONENT ADDER                69
IU
    DATSTP     (2)      CONTROLS DATA FED TO AU (DATA IS ORED WITH         40
```

```
                    OUTPUT OF ALU):
                          0 → DATA FROM MEMORY [CMRD]
                          1 → ALL ZEROS
                          2 → 16-BIT K FIELD OF CURRENT INSTRUCTION
                          3 → P REGISTER
    *NIWCLKTP-(1)     0→HOLD, 1→LOAD NIW                                  78
     CIWSTP   (2)     CIW FUNCTION:                                       38
                          0 → LOAD FROM NIW
                          1 → NEWPARCEL (LEFT SHIFT 15)
                          3 → HOLD
    *MADSTP   (1)     SELECTS INPUT TO MA:  0→LOW BITS OF AC, 1→P REG.    74
    *MADCLKTP-(1)     0→HOLD, 1→LOAD MA                                   73
    *PRGSTP-  (2)     P REGISTER FUNCTION:                                75
                          0 → HOLD
                          1 → INCREMENT
                          2 → DECREMENT
                          3 → LOAD FROM AC
    *CMFCN    (2)     CENTRAL MEMORY FUNCTION:                            79
                          0 → HOLD
                          1 → CLEAR REQUEST
                          2 → ISSUE READ REQUEST
                          3 → ISSUE WRITE REQUEST
     LATCHI   (1)     0→LOAD I LATCH WITH I FIELD OF CURRENT INSTRUCTION 27
                      1→HOLD
MICROPROGRAM ADDRESS CONTROL
     CNDSTP   (6)     SELECTS CONDITION TO BE TESTED                      28
                          0 → FALSE
                          1 → EXPONENT ALU PROPAGATE OUTPUT
                          2 → EALU P OUTPUT ∨ ¬ EALU OPERAND 2 BIT 10
                          3 → EALU P OUTPUT ∨ EALU OPERAND 2 BIT 10
                          4 → EXPONENT ALU BIT 11
                          5 → EXPONENT ALU BIT 0 ∧ BIT 1
                          6 → EXPONENT ALU BIT 2 ∧ BIT 3
                          7 → EXPONENT ALU BIT 4 ∧ BIT 5
                          10 → FOFL
                          11 → XFOFL
                          12 → M (LOW 4 BITS OF MQ) > 7
                          13 → M (LOW 4 BITS OF MQ) > 6
                          14 → EALU BITS 0 THROUGH 3 NOT ALL 0
                          15 → REGISTER OUTPUT BIT 59
                          16 → REGISTER OUTPUT BIT 17
                          17 → BUF BIT 59
                          20 → AC BIT 59
                          21 → AC BIT 49
                          22 → AC BIT 47
                          23 → AC BIT 46
                          24 → ALU OUTPUT BIT 59 ≠ BIT 47
                          25 → ALU PROPAGATE OUTPUT
                          26 → MQ BIT 59
                          27 → MQ BIT 50
                          30 → MQ BIT 49
                          31 → ALU OUTPUT BIT 59
                          32 → ALU OUTPUT BIT 48 ≠ BIT 59
                          33 → BUF BIT 59 ≠ REGISTER OUTPUT BIT 59
```

```
                        34 → ALU GENERATE OUTPUT
                        35 → ALU OUTPUT BIT 49
                        36 → ALU OUTPUT BIT 47
                        37 → AC<<BUF ^ ¬ MQ(49)
                        40 → OPCODE BIT 0
                        41 → OPCODE BIT 1
                        42 → OPCODE BIT 2
                        43 → I LATCH BIT 0
                        44 → I LATCH BIT 1
                        45 → I LATCH BIT 2
                        46 → I LATCH = 0
                        47 → I LATCH > 5
                        50 → CURRENT INSTRUCTION J FIELD = 0
                        51 → MODE2 (EXTERNALLY SET FLAG)
                        52 → MODE4 (EXTERNALLY SET FLAG)
                        53 → CMDONE (SET BY CENTRAL MEMORY)
                        54 → LASTPARCEL
                        55 → NIWEMPTY
                        56 → ICHECK
                        57 → EXTINT (EXTERNALLY SET FLAG)
BRADDR0   (10)    BRANCH ADDRESS IF CONDITION IS FALSE                  11
OPCDBR0   (1)     1→REPLACE LOW 6 BITS OF BRADDR0 BY OPCODE OF          21
                  CURRENT INSTRUCTION
BRADDR1   (10)    BRANCH ADDRESS IF CONDITION IS TRUE                    0
OPCDBR1   (1)     1→REPLACE LOW 6 BITS OF BRADDR1 BY OPCODE OF          11
                  CURRENT INSTRUCTION
SPCLFCN   (1)     1→ENABLE SPECIAL FUNCTIONS (SEE FOOTNOTE)             81
```

NOTE: A MICROINSTRUCTION CONTAINS 85 BITS, NUMBERED 0 TO 84. EACH FIELD OCCUPIES A CONSECUTIVE GROUP OF BITS IN A MICROINSTRUCTION; THE BIT POSITION COLUMN SPECIFIES THE NUMBER OF THE LOWEST-NUMBERED BIT IN THE GROUP.

* FIELDS MARKED WITH AN * ARE SPECIAL FIELDS. THE EFFECTIVE VALUE OF A SPECIAL FIELD IS OBTAINED BY ANDING ITS BITS IN THE MICROINSTRUCTION WITH THE SPECIAL FUNCTION FIELD, -SPCLFCN-. THE SPECIAL FIELDS OCCUPY BIT POSITIONS ALSO USED BY THE CONSTANT FIELD OF THE EXPONENT AU. IF A CONSTANT IS NEEDED IN A MICROINSTRUCTION, SPCLFCN IS SET TO 0 AND ALL SPECIAL FIELDS ARE EFFECTIVELY 0. IF A NON-ZERO SPECIAL FIELD IS REQUIRED IN A MICROINSTRUCTION, SPCLFCN IS SET TO 1 AND THE SPECIAL FIELDS ARE SET TO THEIR REQUIRED VALUES; PRESUMABLY IN THIS CIRCUMSTANCE THE FIELD WILL NOT SIMULTANEOUSLY BE USED AS A CONSTANT BY THE EXPONENT AU, SINCE IT IS MOST UNLIKELY THAT THE BIT PATTERN PRODUCED BY COMBINING THE SPECIAL FIELDS WILL BE A USEFUL CONSTANT.

6. The microprogram for CDC 6600 emulation

Presented below is the complete PUMA microprogram for emulation of the Control Data 6600 central processor, consisting of 454 microinstructions. This microcode was originally developed by the author concurrently with the design of the PUMA, and was subsequently improved by R. Kenner and annotated by A. Czerniakiewicz.

Although the microcode is extensively commented, at least one point deserves separate mention here. When the instructions in one word have been executed, CIW = NIW will be performed to bring the next instruction word into CIW. The microprogram will then immediately issue a read request for the following word to fill NIW, which is now empty. This read operation will proceed concurrently with the execution of the first instructions in the instruction word. The condition NIWEMPTY ∧ CMDONE indicates that memory is ready with data which is to be loaded into NIW. This condition will be checked at the end of the microprogram sequence defining each 6600 operation. Thus the last instruction of each such sequence will include the test

IF ICHECK THEN ICHECK ELSE OPCODEBRANCH

where ICHECK serves both as the mnemonic for the condition LASTPARCEL ∨ (NIWEMPTY ∧ CMDONE) and the label for the code handling the condition. This code sequence appears on the third page of the microcode. If LASTPARCEL is not true (have not executed last instruction in a word), the microprogram will load the next instruction word into NIW and then continue with execution of the current instruction word.

Because the fetch of data for NIW occurs concurrently with instruction execution, every microcode sequence which involves a memory access must check whether this fetch is still going on (whether NIWEMPTY). If it is, the microcode must complete this fetch before beginning the next memory operation. The sequences which perform this check are: the branch sequence (executed whenever a branch is taken);

the return jump (opcode 01); and register load/store (opcodes 50 to 57).

This emulator differs functionally from the CDC 6600 in a few minor respects:

1. Rounded floating-point instructions produce their results by post-rounding (adding 1/2 to the final coefficient). In some circumstances this may produce a more accurate result than the 6600, which uses pre-rounding.

2. Floating-point runs are computed using an effective 110-bit accumulator, whereas the 6600 uses a 98-bit accumulator. In consequence, double-precision sums may differ from their 6600 counterparts by 1 in the low-order bit.

```
*****************************************************************************
*                                                                           *
*          EXCHANGE JUMP                                                    *
*                                                                           *
*          THROUGH AN EXCHANGE JUMP SEQUENCE, THE PERIPHERAL PROCESSOR      *
*          CAN CAUSE THE CENTRAL PROCESSOR TO EXCHANGE THE CURRENT VALUES   *
*          OF THE P, X, A, AND B REGISTERS WITH VALUES STORED IN MEMORY.    *
*          THIS OPERATION IS NORMALLY PERFORMED WHEN SWITCHING THE CONTROL   *
*          OF THE CENTRAL PROCESSOR FROM ONE USER-S JOB TO ANOTHER.         *
*          THE VALUES IN MEMORY ARE STORED IN A 16-WORD BLOCK, AS FOLLOWS:  *
*               WORD 0:                 IN BITS 36 TO 53, P                 *
*                                       IN BITS 18 TO 35, A0                *
*                                       IN BITS 0 TO 17, B0 = 0             *
*               IN WORD I, I=1 TO 7     IN BITS 18 TO 35, A(I)              *
*                                       IN BITS 0 TO 17, B(I)               *
*               IN WORD I, I=8 TO 15    X(I-8)                              *
*                                                                           *
*          THE SEQUENCE OF OPERATIONS INVOLVED IN AN EXCHANGE JUMP ARE:     *
*               1)  THE PP RAISES THE EXTERNAL INTERRUPT (EXTINT) LINE      *
*               2)  THE CP STOPS EXECUTING INSTRUCTIONS AT THE END OF THE   *
*                   CURRENT INSTRUCTION WORD, AND SETS  P=-0 AS A SIGNAL    *
*                   TO THE PP                                               *
*               3)  THE PP PUTS THE ADDRESS OF THE EXCHANGE PACKAGE ONTO     *
*                   THE CMRD LINES.  ON MACHINES WITH AN RA AND FL, THE PP  *
*                   WILL ALSO RESET THESE REGISTERS (SINCE THE EXCHANGE     *
*                   PACKAGE WILL NORMALLY BE OUTSIDE THE USER-S FIELD       *
*                   LENGTH).                                                *
*               4)  THE PP LOWERS THE EXTINT LINE, SIGNALLING THE CP TO     *
*                   BEGIN THE REGISTER EXCHANGE                             *
*               5)  THE CP SETS P=0 TO SIGNAL THE COMPLETION OF THE REGISTER*
*                   EXCHANGE.                                               *
*               6)  THE PP NOW RESETS RA AND FL TO THE REFERENCE ADDRESS     *
*                   AND FIELD LENGTH OF THE NEW USER PROGRAM, AND THEN      *
*                   RAISES AND DROPS LINE EXTINT ONCE MORE TO SIGNAL TO     *
*                   THE CP TO START INSTRUCTION EXECUTION                   *
*                                                                           *
*****************************************************************************
*
*          WHEN THE CP IS POWERED UP, MICROPROGRAM EXECUTION WILL NORMALLY
*          BE INITIATED STARTING WITH MICROINSTRUCTION -WAIT-.
*          E1 IS SET TO ACCESS REGISTER NUMBER ZERO (A, B)
WAIT     AC=CMRD; E1=270;  IF EXTINT THEN WAIT ELSE XJP1    * PICK UP CMRD.
XJP      AC=-0; IF  ¬CMDONE THEN XJP
         MQ=P; P=AC;  CLEAR; GO WAIT
XJP1     MA=AC; P=AC;  AC%MQ=SHIFT(AC%MQ,01); CLEAR
XJP2     IF CMDONE THEN XJP2
*          POINT TO START OF PACKAGE.  NOW BUILD 18 BITS MASK FOR
*          LATER USE IN EXTRACTING REGISTERS FROM PACKAGE.
         READ; AC%MQ=SHIFT(AC%MQ,R1)       * RECREATE MQ.
         AC=SHIFT(AC%MQ,A1); BUF=AC;  E0=12
XJPSL    AC=SHIFT(AC%MQ,A4); E0=E0-1[F]; IF EALU(0-3) THEN XJPSL
*          NOW SAVE MASK AND OLD P-REGISTER VALUE.
         Y0=AC; AC=MQ
         Y1=AC; MQ=0; GO XJPENT
```

```
*
*           THIS IS THE FIRST LOOP OF THE EXCHANGE JUMP CODE.  THIS
*           LOOP INSERTS THE A AND B REGISTERS INTO THE NEW EXCHANGE
*           PACKAGE AND EXTRACTS THEM FROM THE OLD PACKAGE.
XJPL1     AC%MQ=SHIFT(BUF%MQ,R16)
          AC%MQ=SHIFT(AC%MQ,R1)
          AC%MQ=SHIFT(AC%MQ,R1); BUF=AM
XJPENT    AC%MQ=SHIFT(BUF%MQ,R16)
          AC%MQ=SHIFT(AC%MQ,R1)
          AC%MQ=SHIFT(AC%MQ,R1); BUF=Y1            * GET OLD P.
          AC%MQ=SHIFT(BUF%MQ,R16)
          AC%MQ=SHIFT(AC%MQ,R4)
          AC%MQ=SHIFT(AC%MQ,R4)
XJPWT1    AC=MQ;  IF ¬CMDONE THEN XJPWT1
          MQ=CMRD; CLEAR
XJPWT2    IF CMDONE THEN XJPWT2
          WRITE
*           NOW WAIT FOR WRITE ACCEPT.
XJPWT3    IF ¬CMDONE THEN XJPWT3
          CLEAR; AC=MQ; BUF=Y0              * GET MASK.
          Y2=AC; AC=AC^-BUF
          BM=AC; MQ=0                      * STORE B REGISTER.
          BUF=Y2
          AC=SHIFT(BUF%MQ,R16)
          AC=SHIFT(AC%MQ,R1)
          AC=SHIFT(AC%MQ,R1); BUF=Y0
          Y2=AC; AC=AC^-BUF; P=P+1
*           SET -A- REGISTER AND SEE IF MUST EXTRACT P.
          AM=AC; =7+E1; IF EALUPOUT- THEN XJEXTP
          E1=E1+7760[F]; AC=0      * INCREMENT NUMBER AND CLEAR AC
XJPWT4    BUF=BM; IF CMDONE THEN XJPWT4  * WAIT FOR MEMORY.
          Y1=AC; MA=P; READ
          =E1; IF EALU(0-3) THEN XJPL1 ELSE XJPCON
*
*           NOW WE HAVE THE ≠APPENDAGES≠ FOR THE ABOVE ROUTINE.
*           WE HAVE THE CODE TO EXTRACT THE NEW VALUE OF THE P
*           REGISTER FROM THE FIRST WORD OF THE EXCHANGE PACKAGE.  IT IS
*           SAVED IN Y4.
XJEXTP    BUF=Y2; E1=E1+7760[F]
          AC=SHIFT(BUF%MQ,R16)
          AC=SHIFT(AC%MQ,R1)
          AC=SHIFT(AC%MQ,R1); BUF=Y0               * GET MASK.
          AC=AC^-BUF
          Y4=AC; AC=0; GO XJPWT4
*
*           NOW CONTINUE WITH THE MAIN CODE.  NOW GET THE X REGISTERS.
XJPCON    E1=270; B0=AC      * CLEAR B0; RESET REGISTER NUMBER.
XJPLOOP2  BUF=XM; MQ=CMRD; IF ¬CMDONE THEN XJPLOOP2
XJPWT6    AC=BUF; CLEAR; IF CMDONE THEN XJPWT6
          WRITE; P=P+1
XJPWT7    IF ¬CMDONE THEN XJPWT7
          MA=P; AC=MQ; BUF=Y4; CLEAR
          XM=AC; AC=0; E1=E1+7760[F]; IF ¬EALU(0-3) THEN XJPDONE
XJPWT8    IF CMDONE THEN XJPWT8
```

```
         READ; GO XJPLOOP2
XJPDONE  P=AC; IF ¬EXTINT THEN XJPDONE
WAIT1    AC=BUF; IF EXTINT THEN WAIT1 ELSE BR2
*
*****************************************************************************
*                                                                           *
*          ICHECK SEQUENCE%  BRANCH HERE WHEN ALL INSTRUCTIONS IN AN        *
*          INSTRUCTION WORD HAVE BEEN EXECUTED, OR WHEN NEXT INSTRUC-       *
*          TION WORD IS AVAILABLE TO LOAD INTO NIW.                         *
*                                                                           *
*****************************************************************************
*
ICHECK   IF LASTPARCEL THEN ADVANCEP
*          NIWEMPTY ^ CMDONE IS TRUE;  FETCH NEXT INSTRUCTION WORD
*          AND GO TO NEXT INSTRUCTION
         NIW=CMRD;  CLEAR;  LATCH I;  GO OPCODEBRANCH
*          CURRENT INSTRUCTION WORD IS COMPLETE
*          HAS NEXT INSTRUCTION WORD BEEN FETCHED?
ADVANCEP P=P+1;  IF NIWEMPTY THEN RNIWAIT
*          YES, PUT NEXT INSTRUCTION WORD INTO CIW
LOADCIW  CLEAR; CIW=NIW; IF CMDONE THEN LOADCIW
*          INITIATE READ OF NEXT INSTRUCTION WORD AND BRANCH TO
*          EXECUTE FIRST INSTRUCTION OF CURRENT WORD
RNI      MA=P; READ; LATCH I;  IF ¬EXTINT THEN OPCODEBRANCH
*          EXTERNAL INTERRUPT HAS BEEN RAISED, RESET P TO ADDRESS OF NEXT
*          INSTRUCTION TO BE EXECUTED WHEN EXECUTION OF THIS PROGRAM RESUMES,
*          AND ENTER EXCHANGE JUMP SEQUENCE.
         P=P-1; GO XJP
RNIWAIT  NIW=CMRD; IF ¬CMDONE THEN RNIWAIT ELSE LOADCIW
*
*****************************************************************************
*                                                                           *
*          BRANCH SEQUENCE                                                  *
*                                                                           *
*****************************************************************************
*
*          COMPLETE FETCH OF NEXT INSTRUCTION IN SEQUENCE
BRANCH   IF ¬NIWEMPTY THEN BR2
BRWAIT1  NIW=CMRD;  IF ¬CMDONE THEN BRWAIT1
*          EMPTY NIW, RESET P, WAIT FOR CMDONE TO GO LOW
BR2      CLEAR;  CIW=NIW;  P=AC;  IF ¬CMDONE THEN BR3
BRWAIT2  IF CMDONE THEN BRWAIT2
BR3      MA=P;  READ;  P=P+1; GO RNIWAIT
*
*****************************************************************************
*                                                                           *
* ERROR STOP: STORE MODE AND ADDRESS OF CURRENT INSTRUCTION + 1             *
*             IN WORD 0 AND JUMP TO 0.                                      *
*             MODE BIT + 2000 SHOULD BE IN E0 ON ENTRY                      *
*                                                                           *
*      MODES: 0= PROGRAM STOP                                               *
*             2= INFINITE OPERAND                                           *
*             4= INDEFINITE OPERAND                                         *
*                                                                           *
```

```
*****************************************************************************
*
ERROR     AC=P;  E2=36
          MQ=0;  IF ¬NIWEMPTY THEN ERRORLP
ERRWT     IF ¬CMDONE THEN ERRWT
*           ***
*           *  P (ADDRESS OF CURRENT INSTRUCTION + 1) IS SHIFTED LEFT 30 BITS
*           *  AND THEN PACKED WITH E0.  PACK OPERATION STRIPS THE 2000 BIT
*           *  FROM E0, LEAVING MODE BIT AND P IN Y0.
*           ***
ERRORLP   Y0=E0%AC;  AC=SHIFT(AC%MQ,L1);  =E2-1
          E2=E2-1;  IF ¬EALU(11) THEN ERRORLP
ERRSTORE CLEAR;  BUF=Y0; AC=0; IF CMDONE THEN ERRSTORE
          MA=AC; AC=BUF; WRITE
ERRORWLP IF ¬CMDONE THEN ERRORWLP
*           ***
*           *  SET P = 0 AND LOOP WAITING FOR EXTINT (EXTERNAL INTERRUPT)
*           ***
          AC=0
HALT      P=AC; IF EXTINT THEN XJP ELSE HALT
*
*************
*
*           SPECIAL FLOATING POINT RESULTS
*
*************
*
*           ***
*           *  FLRESFLO: EXPONENT OVER/UNDERFLOW - ON ENTRY, EXPONENT IN E0.
*           *        E0<0:  UNDERFLOW, STORE 0 IN XI
*           *        E0>0:  OVERFLOW, STORE INFINITY IN XI
*           *        (FLRSFLON -- SAME PLUS DO NEWPARCEL)
*           ***
FLRSFLON =E0; NEWPARCEL; IF EALU(11) THEN WXIZERO ELSE WXIINF
FLRESFLO =E0;  IF EALU(11) THEN WXIZERO ELSE WXIINF
*           ***
*           *  WXIZERO: ZERO RESULT
*           ***
WXIZERON AC=0; NEWPARCEL; GO WXI
WXIZERO  AC=0
WXI       XI=AC;  LATCH I; IF ICHECK THEN ICHECK ELSE OPCODEBRANCH
*           ***
*           *  WXIINDEF: INDEFINITE RESULT. I.E. EXPONENT OF XI EQUAL 1777.
*           ***
WXIINDEF AC=0;  E0=7777
WXIFLOAT XI=E0%AC;  LATCH I; IF ICHECK THEN ICHECK ELSE OPCODEBRANCH
*           ***
*           *  WXIINF : INFINITE RESULT. SIGN DETERMINED BY SIGN OF AC.
*           *
*           *            TEST AC(59)   CHECKS SIGN OF AC ON ENTRY:
*           *                  IF AC(59)=0 (I.E. POS.),WHEN PACKING:AC= 0 ;E0=1777
*           *                                GIVES EXP. 3777 = PLUS INFINITY.
*           *                  IF AC(59)=1 (I.E. NEG.),WHEN PACKING:AC=-0 ;E0=1777
*           *                                GIVES EXP. 4000 = MINUS INFINITY
```

```
*          ***
WXIINFN   NEWPARCEL; AC=0; MQ=SHIFT(AC%MQ, 01);  E0=1777;
+         IF AC(59) THEN WXIMIINF ELSE WXIFLOAT
WXIINF    AC=0; MQ=SHIFT(AC%MQ, 01); E0=1777;
+         IF AC(59) THEN WXIMIINF ELSE WXIFLOAT
WXIMIINF AC=SHIFT(AC%MQ, R1); GO WXI
*          ***
*          *  INDEFOP: INDEFINITE OPERAND ; ONE (OR BOTH) OF THE EXPONENTS IN
*          *           E1 AND E2 ARE INDEFINITE
*          ***
INDEFOPN E0=2004; NEWPARCEL; IF MODE4 THEN ERROR ELSE INDEFOP2
INDEFOP   E0=2004;  IF MODE4 THEN ERROR
INDEFOP2 IF INF(E1) THEN INFOPTOO
         IF ¬INF(E2) THEN WXIINDEF
INFOPTOO E0=2002;  IF MODE2 THEN ERROR ELSE WXIINDEF
*
*
NEWINSLO NEWPARCEL; IF ICHECK THEN ICHECK
NEWINSTR LATCH I; IF ICHECK THEN ICHECK ELSE OPCODEBRANCH
*
**************
*
*  PS. OP 00 . PROGRAM STOP.
*
**************
00        E0=2000;  GO ERROR     * STORES MODE = 0 IN E0.
*
*****************************************************************************
*                                                                           *
* RJ K : RETURN JUMP.  OP 01 .                                              *
*                                                                           *
*         WHEN A RETURN JUMP OCCURS AT LOCATION <HERE> TO LOCATION <THERE> THE  *
*         FOLLOWING HAPPENS:                                                *
*                                                                           *
*         STEP 1: AT LOCATION <THERE> IS STORED A JUMP TO <HERE + 1 > (I.E.     *
*                 0400 P ) AND                                              *
*         STEP 2: CONTROL TRANSFERS TO <THERE + 1>                          *
*                                                                           *
*****************************************************************************
01        MQ=P;  E0=2400
*           SHIFT P INTO HIGH 30 BITS OF AC
          AC%MQ=SHIFT(0%MQ,R16)
          AC%MQ=SHIFT(AC%MQ,R16)
          AC%MQ=SHIFT(AC%MQ,L1)
          AC%MQ=SHIFT(AC%MQ,L1)
*           STORE  0400 P  TEMPORARILY IN Y0
          Y0=E0%AC; IF ¬NIWEMPTY THEN 01STORE
01WT      IF ¬CMDONE THEN 01WT
01STORE   BUF=Y0; AC=K;  CLEAR; IF CMDONE THEN 01STORE
          P=AC; MA=AC; AC=BUF; WRITE
01WAIT    IF ¬CMDONE THEN 01WAIT
          P=P+1; CIW=NIW; CLEAR; GO BRWAIT2
**************
*
```

```
* JP BI+K : JUMP TO BI+K- OP 02- VALUE BI+K WILL BE COMPUTED IN AC.
*
**************
02        BUF=BI;  AC=K; IF I=0 THEN BRANCH
          =AC+BUF[18]
          AC=AC+BUF[18];  IF NIWEMPTY THEN BRWAIT1 ELSE BR2
*****************************************************************************
* CC XJ,K (CC=ZR,NZ,PL,NG,IR,OR,DF,ID). OP 030-037                           *
*         CONDITION CODES: WHEN XJ SATISFIES SPECIFIED COND., CONTROL TRANSFERS *
*                          TO K.                                             *
*           ZR:CONTROL TRANSFERS TO K IF XJ IS ZERO (BOTH PLUS ZERO AND MINUS *
*              ZERO SATISFY CONDITION).                                       *
*           NZ:BRANCH WHEN XJ CONTAINS ANYTHING OTHER THAN PLUS AND MINUS ZERO *
*           PL:TRANSFER TO K IF XJ IS POSITIVE (I.E.BIT 59 IS ZERO)           *
*           NG:BRANCH IF XJ IS NEGATIVE (I.E.BIT 59 IS 1).                    *
*                                                                             *
*         ILLEGAL EXPONENTS:                                                  *
*           IR:JUMP TO K IF XJ IS IN RANGE (I.E. 12 HIGH BITS OF XJ ARE NOT   *
*              3777 OR 4000.                                                  *
*           OR:JUMP TO K IF XJ IS OUT OF RANGE (I.E. 12 HIGH BITS OF XJ ARE 3777*
*              OR 4000.                                                       *
*           DF:JUMP TO K IF XJ IS DEFINITE (I.E.12 HIGH BITS OF XJ ARE NOT    *
*              1777 OR 6000.                                                  *
*           ID:JUMP TO K IF XJ IS INDEFINITE (I.E.12 HIGH BITS OF XJ ARE  1777 *
*              OR 6000.                                                       *
*                                                                             *
*         CONDITIONS I(X),X=0,1,2,BELOW,REFER TO I FIELD (BITS 21-23) OF OP,AND *
*           WHEN SET WILL INDICATE WHICH OF OP 030-037 IS INDICATED           *
*                                                                             *
*         THUTH TABLE: T=TRUE=1, F=FALSE=0                                    *
*                                                                             *
*              ID   DF   OR   IR   NG   PL   NZ   ZR                          *
*           OP 037  036  035  034  033  032  031  030                         *
*                                                                             *
*          I(2)  T    T    T    T    F    F    F    F                         *
*                                                                             *
*          I(1)  T    T    F    F    T    T    F    F                         *
*                                                                             *
*          I(0)  T    F    T    F    T    F    T    F                         *
*                                                                             *
*****************************************************************************
03        AC=K; IF I(2) THEN FLOATBR      *BRANCH OCCURS FOR OP 034-037
*                                         *IN ANY CASE AC= K
          BUF=XJ; MQ=AC; IF I(1) THEN POSNEGBR *I(1) TRUE MEANS OP 032 AND 031.
*                                              *IN ANY CASE BUF= XJ , MQ= K.
          AC=BUF; NEWPARCEL;  IF I(0) THEN 031
**************
*
* 030: ZR XJ,K .BRANCH WHEN EITHER XJ =+0 OR XJ = -0.
*
**************
          NEWPARCEL; IF AC=0 THEN MQBRANCH
*           ***
*           *  MQBRANCH: WHEN ADDRESS HAS BEEN KEPT WAITING IN MQ, AND IT HAS TO
```

```
*           *             BE TRANSFERED FIRST TO AC.
*           *  IF AC ≠ +0, CHECK FOR -0.
*           ***
          AC=-AC
          IF AC=0 THEN MQBRANCH ELSE NEWINSTR
*           ***
*           *  BOTH +0 AND -0 HAVE BEEN CHECKED AND BRANCH DOES NOT OCCUR.
*           ***
***************
*
* 031: NZ XJ,K .BRANCH WHEN BOTH XJ ≠ +0 AND XJ ≠ -0.
*
***************
031       NEWPARCEL; IF AC=0 THEN NEWINSTR
          AC=-AC
          IF AC=0 THEN NEWINSTR ELSE MQBRANCH
*           ***
*           *  POSNEGBR: CORRESPONDS TO OP 032 AND 033.
*           *            ON ENTRY: BUF = XJ  , MQ = K .
*           ***
POSNEGBR  AC=BUF; NEWPARCEL;    IF I(0) THEN 033
***************
*
* 032: PL XJ,K . BRANCH IF XJ IS POSITIVE.
*
***************
          AC=MQ; NEWPARCEL;    IF ¬AC(59) THEN BRANCH ELSE NEWINSTR
*                               *BRANCHING OCCURS WHEN AC = XJ  IS NON NEGATIVE
*                               *ADDRESS OF BRANCH IS PASSED FROM MQ.
***************
*
* 033: NG XJ,K . BRANCH IF XJ IS NEGATIVE
*
***************
033       AC=MQ; NEWPARCEL;    IF AC(59) THEN BRANCH ELSE NEWINSTR
*           ***
*           *  FLOATBR: OP CODES FOR ILLEGAL OPERANDS, I.E. INFINITE AND
*           *           INDEFINITE OPERANDS.
*           ***
FLOATBR   EO:BUF=XJ;  IF I(1) THEN INDEFBR      * I(1) TRUE MEANS OP 036 AND 037
          NEWPARCEL; IF I(0) THEN 035
***************
*
* 034: IR XJ,K .IN RANGE BRANCH. I.E. BRANCH WHEN 12 BITS OF XJ ARE
*              NOT 3777 OR 4000.
*
***************
          NEWPARCEL; IF ¬INF(EO) THEN BRANCH ELSE NEWINSTR
***************
*
* 035: OR XJ,K . OUT OF RANGE BRANCH. I.E. BRANCH WHEN 12 HIGH BITS OF XJ
*              ARE 3777 OR 4000.
*
***************
```

```
035       NEWPARCEL; IF INF(EQ) THEN BRANCH ELSE NEWINSTR
INDEFBR   NEWPARCEL; IF I(0) THEN 037
***************
*
* C36: DF XJ,K    DEFINITE BRANCH -- BRANCH WHEN 12 HIGH BITS OF XJ
*                 ARE NOT 1777 OR 6000
*
***************
          NEWPARCEL; IF ¬INDEF(EQ) THEN BRANCH ELSE NEWINSTR
***************
*
* C37: ID XJ,K    INDEFINITE BRANCH -- BRANCH WHEN 12 HIGH BITS OF XJ
*                 ARE 1777 OR 6000
*
***************
037       NEWPARCEL; IF INDEF(LO) THEN BRANCH ELSE NEWINSTR
***************
*
* EQ BI,BJ,K - OP 04- BRANCH WHEN BI=BJ ( B REGISTERS HAVE ONLY 18 BITS).
*           SINCE THE CASE OF      EQ X     IS SO COMMON, WE WILL TEST FOR IT TO
*           SAVE 2 (OF THE 4) CYCLES.
*
***************
04        BUF=BI; MQ=K; IF J=0 THEN 04BJZERO
          AC=BUF; BUF=BJ; NEWPARCEL
          AC=AC/BUF    *EXCLUSIVE OR, I.E. AC=0 ONLY WHEN BI AND BJ COINCIDE
*           ***
*           *  AT THIS POINT MQ = K .
*           ***
04TEST    AC=MQ; NEWPARCEL; IF  AC=0 THEN BRANCH ELSE NEWINSTR
*                             * THE TEST AC = 0 REFERS TO PREVIOUS CONTENTS
*                             * OF AC, I.E. BI/BJ.
*                             * IN AC=MQ, AC IS GIVEN ADDRESS OF BRANCH.
04BJZERO  AC=BUF; NEWPARCEL; IF I=0 THEN MQBRANCH ELSE 04TEST
***************
*
* NE BI,BJ,K -OP 05. BRANCH WHEN BI≠BJ.
*
***************
05        BUF=BI;  MQ=K; IF J=0 THEN 05BJZERO
          AC=BUF;  BUF=BJ; NEWPARCEL
          AC=AC/BUF
05TEST    AC=MQ; NEWPARCEL;  IF ¬AC=0 THEN BRANCH ELSE NEWINSTR
05BJZERO  AC=BUF; NEWPARCEL; IF I=0 THEN NEWINSLO ELSE 05TEST
***************
*
* GE BI,BJ,K -OP 06- BRANCH WHEN BI IS GREATER THAN OR EQUAL TO BJ.
*
***************
06        BUF=BI;  MQ=K;  IF REG(17) THEN 06BINEG
*                             * REG(17) TRUE MEANS BI(ONLY 18 BITS) IS NEGATIVE.
          AC=BUF;  BUF=BJ; NEWPARCEL;  IF REG(17) THEN MQBRANCH
*                             * BRANCHING OCCURS BECAUSE BJ IS NEGATIVE, BI IS
*                             * POSITIVE, HENCE BI ≥ BJ.
```

```
*                              * IT IS MQBRANCH BECAUSE WE NEED ADDRESS OF BRANCH
*                              * FROM MQ.
*          ***
*          *  AT THIS POINT:AC= BI , BUF= BJ , MQ= K .
*          *  06SUBTR IS DONE WHEN EITHER BOTH ARE POSITIVE OR BOTH ARE NEGATIVE
*          ***
06SUBTR   =AC-BUF[18]; NEWPARCEL;  IF NIWEMPTY THEN 06EMPTY
          AC=MQ;   =AC-BUF[18];  IF ALU(59) THEN NEWINSTR ELSE BR2
06EMPTY   AC=MQ;   =AC-BUF[18];  IF ALU(59) THEN NEWINSTR ELSE BRWAIT1
*                              * ALU(59) TRUE MEANS BJ > BI.
*          ***
*          *  AT THIS POINT: BUF = BI , MQ= K .
*          *  06BINEG : WE GET HERE WHEN BI IS NEGATIVE.
*          ***
06BINEG   AC=BUF;  BUF=BJ; NEWPARCEL;  IF REG(17) THEN 06SUBTR ELSE NEWINSLO
*                              * WHEN BJ < 0, SINCE BI <0 , SUBTRACT THEM, ELSE
*                              * WHEN BJ ≥ 0, BRANCH DOES NOT OCCUR.
*************
*
* LT BI,BJ,K - OP 07 - INSTRUCTIONS ARE SIMILAR TO OP 06.
*
*************
07        BUF=BI;   MQ=K;   IF REG(17) THEN 07BINEG
          AC=BUF;   BUF=BJ; NEWPARCEL;  IF ¬REG(17) THEN 07SUBTR ELSE NEWINSLO
07SUBTR   =AC-BUF[18]; NEWPARCEL;  IF NIWEMPTY THEN 07EMPTY
          AC=MQ;   =AC-BUF[18];  IF ALU(59) THEN BR2 ELSE NEWINSTR
07EMPTY   AC=MQ;   =AC-BUF[18];  IF ALU(59) THEN BRWAIT1 ELSE NEWINSTR
07BINEG   AC=BUF;  BUF=BJ; NEWPARCEL;  IF ¬REG(17) THEN MQBRANCH ELSE 07SUBTR
*
MQBRANCH AC=MQ; IF NIWEMPTY THEN BRWAIT1 ELSE BR2
*
*****************************************************************************
*                                                                           *
* BOOLEAN INSTRUCTIONS                                                      *
*                                                                           *
*      OP 10-  BXI XJ     - TRANSFER A 60-BIT WORD FROM XJ TO XI.           *
*      OP 11-  BXI XJ*XK  - LOGICAL PRODUCT OF XJ AND XK IS PASSED TO X1-(BIT *
*                           IN XI IS 1 WHEN CORRESPONDING BITS IN BOTH XJ AND *
*                           XK ARE 1).                                      *
*      OP 12-  BXI XJ+XK  - LOGICAL SUM OF XJ AND XK TO XI-(BIT IN XI IS 1 WHEN *
*                           CORRESPONDING BIT IN EITHER XJ OR XK IS 1).     *
*      OP 13-  BXI XJ-XK  - LOGICAL DIFFERENCE (EXCLUSIVE OR) OF XJ AND XK TO XI*
*                           (BIT IN XI IS 1 WHEN CORRESPONDING BITS IN XJ AND *
*                            XK ARE UNLIKE)                                 *
*      OP 14-  BXI -XK    - TRANSMIT THE COMPLEMENT OF XK TO XI-            *
*      OP 15-  BXI -XK*XJ- - THE LOGICAL PRODUCT OF XJ AND THE COMPLEMENT OF *
*                           XK IS  PASSED TO XI                             *
*      OP 16-  BXI -XK+XJ- - THE LOGICAL SUM OF XJ AND THE COMPLEMENT OF XK IS *
*                           PASSED TO X1.                                   *
*      OP 17-  BXI -XK-XJ- - THE LOGICAL DIFFERENCE OF XJ AND THE COMPLEMENT OF*
*                           XK TO X1.                                       *
*                                                                           *
*****************************************************************************
*
```

```
*           BXI    XJ    (3 CYCLES)
10        BUF=XJ
WX1BUF    AC=BUF;  NEWPARCEL; GO WXI
*           BXI    XJ*XK    (4 CYCLES)
11        BUF=XK
          AC=BUF;  BUF=XJ
AND       AC=AC^BUF;  NEWPARCEL; GO WXI
*           BXI    XJ+XK    (4 CYCLES)
12        BUF=XK
          AC=BUF;  BUF=XJ
OR        AC=AC\BUF;  NEWPARCEL; GO WX1
*           BXI    XJ-XK    (4 CYCLES)
13        BUF=XK
          AC=BUF;  BUF=XJ
EXOR      AC=AC/BUF;  NEWPARCEL; GO WXI
*           BXI    -XK    (3 CYCLES)
14        BUF=XK
          AC=-BUF;  NEWPARCEL; GO WX1
*           BXI    -XK*XJ    (4 CYCLES)
15        BUF=XK
          AC=-BUF;  BUF=XJ;  GO AND
*           BXI    -XK+XJ    (4 CYCLES)
16        BUF=XK
          AC=-BUF;  BUF=XJ;  GO OR
*           BXI    -XK-XJ    (4 CYCLES)
17        BUF=XK
          AC=-BUF;  BUF=XJ;  GO EXOR
*
*****************************************************************************
*                                                                           *
* LXI JX - OP 20 - LEFT CIRCULAR SHIFT XI, JK PLACES.                       *
*                                                                           *
*          -SINCE ONLY  BIG RIGHT CIRCULAR SHIFTS ARE PERFORMED BY THE MACHINE, *
*           JK-LEFT IS FIRST TRANSFORMED INTO A (74(OCTAL)-JK)-RIGHT SHIFT,OR, *
*           WHEN JK>60 INTO (120 - JK)-RIGHT SHIFT. EO CARRIES THE RIGHT SHIFT *
*           COUNT.                                                          *
*          -MQ AND AC WILL BOTH CONTAIN XI.                                 *
*          -SHIFTING (0 UNITS) IS DONE AT THE SAME TIME THAT THE EXPONENT IS *
*           BEING REDUCED (BY 0 UNITS), THEREFORE,CALLING N THE NUMBER OF RIGHT *
*           SHIFTS NEEDED, N WILL BE REDUCED BY 16, 4 OR 1 UNTIL IT BECOMES ZERO*
*           ACCORDING TO THE FOLLOWING ALGORITHM:                           *
*                                                                           *
*                IF (N < 16) GO TO TRY4                                     *
*      SHIFT16   N = N - 16                                                 *
*                SHIFT RIGHT 16                                             *
*                IF (N ≥ 16) GO TO SHIFT16                                  *
*      TRY4      IF (N < 4) GO TO TRY1                                      *
*      SHIFT4    N = N - 4                                                  *
*                SHIFT RIGHT 4                                              *
*                IF (N ≥ 4) GO TO SHIFT4                                    *
*      TRY1      IF N =0 GO TO WRITEX1                                      *
*      SHIFT1    N = N - 1                                                  *
*                SHIFT RIGHT 1                                              *
*                IF (N > 0 ) GO TO SHIFT1                                   *
```

```
*     WRITEXI  END                                                              *
*                                                                               *
*********************************************************************************
*
20        BUF=XI;   =74-JK
          AC=BUF;   E0=74-JK;  NEWPARCEL;  IF EALU(11) THEN BIGSHIFT
*                               *IF EALU(11) IS SET, NUMBER OF SHIFTS WAS BIGGER
*                               *THAN 60 HENCE WE WILL SHIFT RIGHT (120 - JK).
*           ***
*           *  AT THIS POINT AC = XI, E0 = 60-JK, AND WE ARE READY
*           *  TO START THE SHIFT.
*           ***
LSHIFT    MQ=AC;   =E0;   IF ¬(EALU(4)∨EALU(5)) THEN 20TRY4
*                               * TRUE MEANS EALU(4)=EALU(5)=0,I.E.0≤E0 < 16
20SH16    AC%MQ=SHIFT(AC%MQ,R16);  E0=E0-20[F];  IF EALU(4)∨EALU(5)  THEN 20SH16
*                               * I.E. IF EITHER EALU(4)=1 OR EALU(5)=1, E0 ≥ 16
*                               * AND WE CAN STILL REDUCE BY 16 = 20 (OCTAL).
20TRY4    =E0;   IF ¬(EALU(2)∨EALU(3)) THEN 20TRY1
*                               * IF BOTH EALU(2)=EALU(3)= 0, 0 ≤ E0 < 4
20SH4     AC%MQ=SHIFT(AC%MQ,R4);   E0=E0-4[F];   IF EALU(2)∨EALU(3) THEN 20SH4
*                               * GO ON SHIFTING BY 4 WHENEVER EALU(2)=1 OR
*                               * EALU(3)=1, I.E.  4 ≤ E0 < 16 .
20TRY1    =E0;   IF ¬(EALU(0)∨EALU(1)) THEN WXI
*                               * IF BOTH EALU(0)=EALU(1)=0, N=0 AND WE ARE DONE.
20SH1     AC%MQ=SHIFT(AC%MQ,R1);E0=E0-1[F];IF EALU(0)∨EALU(1) THEN 20SH1 ELSE WXI
*           ***
*           *  BIGSHIFT: WHEN JK>60, SET E0=120- JK
*           ***
BIGSHIFT  =E0+74;   MQ=AC
          E0=E0+74; IF ¬(EALU(4)∨EALU(5)) THEN 20TRY4 ELSE 20SH16
*
*********************************************************************************
*                                                                               *
* AXI JK - OP 21 - ARITHMETIC SHIFT RIGHT XI, JK PLACES.(I.E. WITH SIGN EXT.)   *
*                                                                               *
*          -PROCEDURE IS SIMILAR TO OP 20, BUT SIMPLER BECAUSE E0 = JK NOW      *
*           GIVES RIGHT SHIFTS.                                                 *
*          -CONTENTS OF MQ NOW DO NOT MATTER BECAUSE THE SHIFT IS RIGHT WITH    *
*           SIGN EXTENSION.                                                     *
*                                                                               *
*********************************************************************************
*
21        BUF=XI; E0=JK; NEWPARCEL
RSHIFT    AC=BUF;   =E0;   IF ¬(EALU(4)∨EALU(5)) THEN 21TRY4
21SH16    AC%MQ=SHIFT(AC%MQ,A16);  E0=E0-20[F];  IF EALU(4)∨EALU(5) THEN 21SH16
21TRY4    =E0;   IF ¬(EALU(2)∨EALU(3)) THEN 21TRY1
21SH4     AC%MQ=SHIFT(AC%MQ,A4);   E0=E0-4[F];   IF EALU(2)∨EALU(3) THEN 21SH4
21TRY1    =E0;   IF ¬(EALU(0)∨EALU(1)) THEN WXI
21SH1     AC%MQ=SHIFT(AC%MQ,A1);E0=E0-1[F];IF EALU(0)∨EALU(1) THEN 21SH1 ELSE WXI
*
*********************************************************************************
*                                                                               *
* LXI BJ,XK - OP 22 - LEFT CIRCULAR SHIFT XK NOMINALLY BJ PLACES TO XI.         *
*                                                                               *
```

```
* AXI BJ,XK -OP 23- ARITHMETIC RIGHT SHIFT XK NOMINALLY BJ PLACES TO XI.       *
*                                                                              *
*        -IF BJ IS POSITIVE THESE INSTRUCTIONS ACT JUST LIKE THE SHIFTS OF     *
*         OP 20 AND 21, WITH THE LOW SIX BITS OF BJ TAKEN AS SHIFT COUNT.      *
*        -IF ANY OF THE BITS 6 THROUGH 10 OF BJ ARE NON-ZERO, THE NOMINAL      *
*         RIGHT SHIFT WILL,INSTEAD OF PERFORMING THE SHIFT, SET XK TO ZERO.    *
*        -IF BJ IS NEGATIVE, EACH INSTRUCTION ACTS AS THE OTHER WOULD WITH     *
*         THE COMPLEMENT OF BJ.                                                *
*                                                                              *
********************************************************************************
*
********** LXI    BJ,XK
22       BUF=BJ;  EO=77;  IF REG(17) THEN 22RIGHT
*                          * WHEN BJ <0, IT IS A RIGHT SHIFT
*                          * 77(OCTAL) FORMS A MASK IN EO THAT WILL GUARANTEE
*                          * THAT AC CONTAIN EXACTLY 6 LOW BITS OF BJ.
         AC=EO
         AC=AC^BUF
22LSHIFT =74-AC;  BUF=XK; IF AC=0 THEN WXIBUF
         EO=74-AC;  AC=BUF;  NEWPARCEL; IF EALU(11) THEN BIGSHIFT ELSE LSHIFT
*          ***
*          *  AT THIS POINT BUF = BJ  IS NEGATIVE
*          ***
22RIGHT  AC=-BUF
22RSHIFT EO=AC;  BUF=XK; IF AC=0 THEN WXIBUF
         AC=0; NEWPARCEL; IF EO(6-10)=0 THEN RSHIFT ELSE WXI
*                          * SINCE BJ <0 , EO= -BJ . IF ANY OF BITS 6-10
*                          * OF EO ARE NONZERO, THIS BEING A RIGHT SHIFT,
*                          * IT RETURNS ZERO.
*
********** AXI    BJ,XK
*
23       BUF=BJ;  EO=77;  IF REG(17) THEN 23LEFT
         AC=BUF;  GO 22RSHIFT
*          ***
*          *  AT THIS POINT BUF = BJ < 0  , EO = 77(OCTAL) CONTAINS A MASK
*          ***
23LEFT   AC=EO
         AC=AC^-BUF;  GO 22LSHIFT
*
********************************************************************************
*                                                                              *
* NXI BJ,XK . OP 24 . NORMALIZE XK INTO XI AND BJ.                             *
*                                                                              *
*                  -XK IS SHIFTED LEFT BIT BY BIT UNTIL THE MOST SIGNIFICANT   *
*                   BIT IS IN BIT 47 .POSITIONS VACATED ON THE RIGHT ARE FILLED*
*                   WITH ZEROS (BINARY ONES IF THE NUMBER IS NEGATIVE).        *
*                  -FOR EACH BIT THAT THE COEFFICIENT IS SHIFTED,THE EXPONENT  *
*                   IS DECREMENTED BY ONE.                                     *
*                  -THE NORMALIZED NUMBER IS PUT IN THE XI REGISTER, AND THE   *
*                   NUMBER OF SHIFTS REQUIRED FOR NORMALIZATION  IS LEFT IN BJ *
*                                                                              *
*         NORMALIZING A ZERO COEFFICIENT ENDS WITH A SHIFT COUNT               *
*         BJ = 48(DECIMAL), AND XI CLEARED TO ZERO.                            *
```

```
*
* ZXI BJ,XK - OP 25 -ROUND AND NORMALIZE XK INTO XI AND BJ.
*
*                  -BEFORE NORMALIZING, THIS INSTRUCTION ATTACHES A 1 BIT
*                   TO THE RIGHT OF THE BINARY POINT.
*
*********************************************************************************
*
24        EO%BUF=XK;  MQ=0;  IF REG(59) THEN NORMNEG
*                            * IF REG(59)=1, XK < 0.
          AC=BUF;  IF ILL(EO) THEN 24ILL ELSE NORMZT
*                            * ILL(EO) MEANS INFINITE OR INDEFINITE EXPONENT
NORMNEG   AC=-BUF;  IF ILL(EO) THEN 24ILL ELSE NORMZT
*           ***
*           *  AT THIS POINT AC = ABS (XK)  , EO = EXP(XK) ,BUF= XK , MQ = 0
*           *  NORMZT CHECKS WHETHER XK = 0.
*           ***
NORMZT    E2=60; IF AC=0 THEN NORMWX12
*           ***
*           *  NOSHTEST CHECKS WHETHER XK IS ALREADY NORMALIZED
*           *           IF NOT, E2 STARTS SHIFT COUNT.
*           *  NORMLOOP WILL KEEP SHIFTING 1 LEFT UNTIL AC(46)=1 : TEST CHECKS
*           *           WHETHER NUMBER WILL BE NORMALIZED (I.E. AC(47)=1) AFTER
*           *           THE PRESENT SHIFT
*           ***
NOSHTEST  E2=0; IF AC(47) THEN 24SHFTDN
NORMLOOP  AC%MQ=SHIFT(AC%MQ,L1);  =E2+2;  IF AC(46) THEN 24PLUS1
*                            * IF AC(46)=1 WE STILL NEED TO INCREASE E2 BY 1
          AC%MQ=SHIFT(AC%MQ,L1);  E2=E2+2;  IF ¬AC(46) THEN NORMLOOP
*                            * SHIFT COUNT IS INCREASED BY 2.
*           ***
*           *  24SHFTDN : WE HAVE FINISHED
*           ***
24SHFTDN  =EO-E2;  IF BUF(59) THEN 24RECOMP
*                            * PROCEDURE NORMALIZES ABS(XK) - SINCE BUF = XK ,
*                            * WHEN NEGATIVE, WE NEED TO COMPLEMENT THE RESULT
          EO=EO-E2;  IF FOFL THEN NORMOFLO  * EO REPRESENTS NEW EXPONENT
*                                           * FOFL CHECKS WHETHER NEW EXPONENT
*                                           * IS OUT OF RANGE.
*           ***
*           *  NORMWX1 :SHIFT IS FINISHED AND WE STORE THE RESULT
*           *           WHEN J=0, BJ = BO = 0  ALWAYS -
*           *           WHEN J≠0  WE LEAVE NUMBER OF SHIFTS IN BJ.
*           ***
NORMWXI   XI=EO%AC;  IF J=0 THEN NEWINSLO
24WBJ     EO=E2
24WBJ2    AC=EO
          BJ=AC; NEWPARCEL;  GO NEWINSTR
*           ***
*           *  NEXT INSTRUCTIONS REPRESENT  SPECIAL TESTS
*           ***
*           ***
*           *  24RECOMP :USED WHEN XK < 0 AND WE ARE DONE NORMALIZING ABS(XK).
*           ***
```

```
24RECOMP AC=-AC;  E0=E0-E2;  IF FOFL     THEN NORMUFLO ELSE NORMWXI
*          ***
*          *  NORMUFLO : USED WHEN THERE IS AN EXPONENT UNDERFLOW
*          *               WHEN SUBTRACTING E0-E2 . LEAVES X1=0.
*          ***
NORMUFLO AC=0
NORMWXI2 E0=E2;  X1=AC; IF J=0 THEN NEWINSLO ELSE 24WBJ2
*          ***
*          *  24PLUS1 : USED WHEN THE FINAL NUMBER OF SHIFTS IS ODD,BECAUSE THE
*          *            NORMALIZE LOOP INCREASES SHIFT COUNT BY 2.
*          ***
24PLUS1  =E2+1
         E2=E2+1;  GO 24SHFTDN
*          ***
*          *  241LL :USED WHEN EXP(XK) IS ILLEGAL-
*          *          FIRST AC =BUF = XK   IS RESET, SO INSTRUCTION CAN BE USED
*          *          BOTH FOR POSITIVE AND NEGATIVE XK.
*          *         -SHIFT COUNT IS SET TO 0, AND XK, UNTOUCHED,IS LEFT IN XI
*          ***
241LL    AC=BUF; E2=0;  GO NORMWXI
*
********** ZXI   BJ,XK
*
*          ***
*          *       FOLLOWING 2 INSTRUCTIONS SET HIGH BIT OF MQ
*          ***
25       MQ=0;  E0=10
         MQ=SHIFT(E0%MQ,R4);  E0%BUF=XK;  IF REG(59) THEN ZNORMNEG
         AC=BUF;  IF ILL(E0) THEN 241LL ELSE NOSHTEST
ZNORMNEG AC=-BUF;  IF ILL(E0) THEN 241LL ELSE NOSHTEST
*
*****************************************************************************
*                                                                           *
* UXI BJ,XK - OP 26 - UNPACK XK TO X1 AND BJ                                *
*                  -COEFFICIENT OF X1 IS LEFT IN XK (WITH SIGN EXTENSION) AND *
*                   THE UNBIASED EXPONENT IN BJ.                            *
*                                                                           *
* PXI BJ,XK - OP 27 - PACK X1 FROM XK AND BJ-                               *
*                  -INSTRUCTION IS THE CONVERSE OF OP 26.                   *
*                                                                           *
*****************************************************************************
*
********** UXI   BJ,XK
26       E0%BUF=XK;  IF J=0 THEN WXIBUF
         AC=E0
         BJ=AC;  AC=BUF;  NEWPARCEL;  GO WXI
*
********** PXI   BJ,XK
*
27       BUF=BJ
         AC=BUF;  BUF=XK; NEWPARCEL
         E0=AC;  AC=BUF;  GO WXIFLOAT
*****************************************************************************
*                                                                           *
```

```
* FLOATING POINT ADDITION AND SUBTRACTION                                      *
*                                                                              *
*       A FLOATING POINT ADD OF TWO 60 BIT NUMBERS INVOLVES THE FOLLOWING:     *
*                                                                              *
*       STEP 1: THE NUMBER WITH THE SMALLER EXPONENT IS CHANGED BY             *
*               SHIFTING IT, SO THAT BOTH NUMBERS HAVE THE SAME EXPONENT.      *
*                                                                              *
*               AFTER SHIFTING, THE SMALLER ADDEND IS A 96 BIT NUMBER.         *
*                                                                              *
*       STEP2 : COEFFICIENTS ARE ADDED, GIVING A RESULT WITH DOUBLE            *
*               PRECISION ACCURACY                                             *
*                                                                              *
*       STEP3 : NORMALIZE THE RESULT IF OVERFLOW.                              *
*                                                                              *
* FXI XJ+XK - OP 30 - FLOATING SUM OF XJ AND XK TO XI.                         *
*                                                                              *
* FXI XJ-XK - OP 31 - FLOATING DIFFERENCE OF XJ AND XK TO XI.                  *
*                                                                              *
*                   -BOTH FLOATING SUM AND DIFFERENCE GIVE THE MOST SIGNIFICANT *
*                    48 BITS OF THE RESULT.                                    *
*                                                                              *
* DXI XJ+XK - OP 32 - FLOATING DOUBLE PRECISION SUM OF XJ AND XK TO XI         *
*                                                                              *
* DXI XJ-XK - OP 33 - FLOATING DOUBLE PRECISION DIFFERENCE OF XJ AND XK TO XI  *
*                                                                              *
*                   -DOUBLE PRECISION SUM AND DIFFERENCE GIVE THE LOW ORDER     *
*                    48 BITS OF RESULT. EXPONENT HAS TO BE REDUCED BY 48       *
*                    FROM EXPONENT FOR FLOATING SUM/DIFFERENCE.                *
*                                                                              *
* RXI XJ+XK -OP 34 - ROUND FLOATING SUM OF XJ AND XK TO XI.                    *
*                                                                              *
* RXI XJ-XK -OP 35 - ROUND FLOATING DIFFERENCE OF XJ AND XK TO XI              *
*                                                                              *
*                   -GIVES THE MOST SIGNIFICANT 48 BITS OF THE RESULT,         *
*                    AFTER ROUNDING                                            *
*                                                                              *
********************************************************************************
*
*
********** FXI    XJ+XK
30        E0%BUF=XK
30A       AC=BUF;  E1%BUF=XJ;  IF ILL(E0) THEN 30ILLEXP
*                              * ILL(E0) TRUE MEANS EXP(XK) IS ILLEGAL.
30B       MQ=BUF;  Y0=AC;  IF ILL(E1) THEN 30ILLEXP
*                              * ILL(E0) TRUE MEANS EXP(XJ) IS ILLEGAL.
          AC=MQ;  MQ=AC;  =E0-E1
*           ***
*           *  AT THIS POINT Y0=MQ=BUF= XK , AC= XJ, E0 =EXP(XK), E1 =EXP(XJ).
*           *  NEXT INSTRUCTION COMPARES THE EXPONENTS AND , IF NECESSARY (I.E.
*           *  GOING TO 30XKSMAL) INTERCHANGES XK AND XJ SO THAT BEFORE BEGINNING
*           *  30LOADBUF, Y0 AND MQ CONTAIN NUMBER WITH HIGHER EXPONENT AND
*           *  AC CONTAINS NUMBER WITH SMALLER EXPONENT
*           ***
          E2=E0-E1;  IF EALU(11) THEN 30XKSMAL
```

```
        BUF=Y0; MQ=0;  IF AC(59) THEN 30NEGAC
*                           * NUMBER WITH SMALLER EXPONENT MAY BE NEGATIVE.
*         ***
*         *  E2 CONTAINS THE DIFFERENCE BETWEEN EXPONENTS,SO IT COUNTS THE
*         *     NUMBER OF SHIFTS REQUIRED TO ALIGN THE COEFFICIENTS.
*         *  TINYTEST: ¬E2(7-11)=0  MEANS E2 ≥ 128 . IN THIS CASE THE
*         *            DIFFERENCE IN EXPONENTS IS TOO BIG FOR BOTH FLOATING
*         *            AND DP.SUM SO NUMBER WITH SMALLER EXPONENT IS LIKE 0.
*         *  THE SIX INSTRUCTIONS FOLLOWING TINYTEST PERFORM THE SHIFT. THEY
*         *            ARE THE SAME USED IN OP 21 SINCE AC:MQ CONTAIN THE
*         *            NUMBER TO BE SHIFTED AND E2 CONTAINS THE SHIFT COUNT.
*         ***
TINYTEST IF ¬E2(6-11)=0 THEN 30ADDSML
30SHIFT  =E2;  IF ¬EALU(4)∨EALU(5) THEN 30TRY4
30SH16   AC%MQ=SHIFT(AC%MQ,A16);  E2=E2-20[F];  IF EALU(4)∨EALU(5) THEN 30SH16
30TRY4   =E2;  IF ¬EALU(2)∨EALU(3) THEN 30TRY1
30SH4    AC%MQ=SHIFT(AC%MQ,A4);  E2=E2-4[F];  IF EALU(2)∨EALU(3) THEN 30SH4
30TRY1   =E2;  IF ¬EALU(0)∨EALU(1) THEN 30SHFTDN
30SH1    AC%MQ=SHIFT(AC%MQ,A1);  E2=E2-1[F];  IF EALU(0)∨EALU(1) THEN 30SH1
*         ***
*         *  AT THIS POINT, THE SMALLER ADDEND IS IN AC:MQ, PROPERLY SHIFTED
*         *  AND BUF CONTAINS THE BIGGER ADDEND.
*         *  SINCE WE ARE ADDING TWO 120-BITS NUMBERS, WE NEED TO SET THE
*         *  PG TO THE PROPAGATE AND CARRY VALUES OF THE LOWER SUM (I.E. OF
*         *  MQ + 0 WHEN BUF ≥ 0 ,OR OF  MQ-0  WHEN BUF IS NEGATIVE).THIS IS
*         *  DONE BY INSTRUCTIONS THROUGH 30ADD, WHICH SET PG BUT DO NOT
*         *  CHANGE THE REGISTERS.
*         ***
30SHFTDN AC=MQ;  MQ=AC;  IF BUF(59) THEN 30NEGBUF ELSE 30POSBUF
30NEGBUF (AC)=AC-0[SAVEPG];  GO 30ADD
30POSBUF (AC)=AC+0[SAVEPG]
30ADD    AC=MQ;  MQ=AC;  IF OPCODE(2) THEN 30DP2
*                           * OPCODE(2)=1 GIVES OP 34 THROUGH 37
*         * WE NOW COMPUTE THE HIGHER SUM USING THE P AND G VALUES JUST SAVED
        =AC+BUF[USEPG];  IF OPCODE(1) THEN 30DP
*                           * OPCODE(1)=1 GIVES OP 32 AND 33
        NEWPARCEL; AC=AC+BUF[USEPG];  =E0+1;  IF ¬ALU(59)/ALU(48) THEN WXIFLOAT
*                           * AFTER ADDITION  ALU(48)≠ ALU(59) (I.E.
*                           * ALU(59)/ALU(48)=1) IF AND ONLY IF OVERFLOW HAS
*                           * HAPPENED.
*         ***
*         *  30DFLO : THERE HAS BEEN AN OVERFLOW FROM COEFFICIENT TO EXPONENT
*         *           AND WE NEED TO SHIFT RESULT 1 RIGHT (WITH SIGN EXTENSION)
*         *           AND CORRECT VALUE OF EXPONENT.
*         * WXIFLOAT: STORES FLOATING RESULT IN XI.
*         ***
        AC=SHIFT(AC%MQ,A1);  E0=E0+1;  GO WXIFLOAT
*         AT THIS POINT, E2 CONTAINS THE DIFFERENCE OF EXPONENTS,
*         WHICH AS BEEN DETERMINED TO BE >= 64.  AC HAS NUMBER
*         WITH SMALLER EXPONENT.  MQ IS FILLED WITH SIGN BIT OF AC.
30ADDSML MQ=AC; AC=MQ; IF ¬E2(7-11) = 0 THEN 30ADDZRO
*         127 >= E2 >= 64; SHIFT COEFFICIENT RIGHT 64 AND THEN CONTINUE
*         IN MAIN SHIFT SEQUENCE.
        MQ=SHIFT(AC%MQ, A4); = E2; IF EALU(4)∨EALU(5) THEN 30SH16 ELSE 30TRY4
```

```
*           E2 >= 128; FILL AC AND MQ WITH SIGN BIT.
30ADDZRO MQ=AC; IF BUF(59) THEN 30NEGBUF ELSE 30POSBUF
*           ***
*           *  30NEGAC : USED WHEN SMALLER ADDEND IS NEGATIVE
*           *            WHEN E2(7-11)≠0, THE SMALLER ADDEND IS LIKE -0.SET AC=-0
*           ***
30NEGAC  MQ=-0; IF E2(6-11)=0 THEN 30SHIFT ELSE 30ADDSML
*           ***
*           *  30XKSMALL: XK IS SMALLER ADDEND AND WE INTERCHANGE OPERANDS.
*           *            ON ENTRY YO=MQ= XK, AC=XJ, EO = EXP (XK), E1= EXP(XJ)
*           *                      E2 =  EXP(XK) - EXP(XJ) < 0
*           ***
30XKSMAL YO=AC;  AC=MQ;  MQ=AC;  E2=7777-E2[F]
         EO=E1; BUF=YO; MQ=0; IF AC(59) THEN 30NEGAC ELSE TINYTEST
*           ***
*           *  30ILLEXP : WHEN ONE OR BOTH EXPONENTS ARE ILLEGAL
*           *           . ON ENTRY: AC= XK , BUF= XJ, EO = EXP(XK), E1= EXP(XJ).
*           *             WHEN EXIT OCCURS BECAUSE OF AN INFINITE RESULT, AC(59)
*           *             CONTAINS THE SIGN OF RESULT (PLUS OR MINUS INFINITY).
*           *
*           *          ALGORITHM :
*           *
*           *              IF (XK = INDEF) GO TO INDEFOP
*           *              IF (XJ = INDEF) GO TO INDEFOP
*           *           *     IF WE GET HERE EITHER XK OR XJ(OR BOTH) ARE INFINITY
*           *              IF (XK ≠ INF) GO TO WXIINF
*           *           *     IF WE GET HERE XK IS INFINITY
*           *            IF (XJ ≠ INF) GO TO WXIINF
*           *           *     IF WE GET HERE BOTH XJ AND XK ARE INFINITY
*           *              LET AC = SIGN ( XK / XJ )     * EXCLUSIVE OR
*           *              IF AC = 1  GO TO INDEFOP
*           *              GO TO WXIINF.
*           ***
30ILLEXP E2=EO
         IF INDEF(EO) THEN INDEFOPN
         MQ=BUF;  IF INDEF(E1) THEN INDEFOPN
*                * AT THIS POINT, EXP(XK) IS IN EO, COEF(XK) IN AC
*                *                EXP(XJ) IS IN E1, COEF(XJ) IN BUF AND MQ
         NEWPARCEL; IF MODE2 THEN INFOPTOO
         AC=MQ;  MQ=AC;  IF ¬INF(EO) THEN WXIINF
         AC=MQ;  MQ=AC;  IF ¬INF(E1) THEN WXIINF
         =AC/BUF;  IF ALU(59) THEN WXIINDEF ELSE WXIINF
*           ***
*           *  FOLLOWING CODE USED FOR DP AND ROUNDED SUM TO COMPUTE FULL
*           *  96 BIT SUM.
*           *  ON ENTRY AC:MQ = SMALLER ADDEND,SHIFTED  ; BUF = BIGGER ADDEND
*           *             EO = EXP OF SUM ;PG SET TO VALUES OF LOWER SUM.
*           ***
30DP2    =AC+BUF[USEPG]
30DP     MQ=AC+BUF[USEPG][SAVEPG]; AC=MQ;  IF BUF(59) THEN 30NEGBF2
*                          * AT THIS POINT AC= LOW BYTE OF SMALLER ADDEND,
*                          * MQ = HIGH BYTE OF SUM ; PG SET TO VALUES OF
*                          * UPPER SUM.
         (AC)=AC+0[USEPG]
```

```
          MQ=AC+0[USEPG];  AC=MQ;  GO 30TOFLO2
*           ***
*           *  30NEGBF2 : IS THE EQUIVALENT OF  THE PREVIOUS 2 INSTRUCTIONS
*           *             WHEN BUF IS NEGATIVE.
*           ***
30NEGBF2  (AC)=AC-0[USEPG]
          MQ=AC-0[USEPG];  AC=MQ
*           ***
*           *  30TOFLO2 : SUM IS DONE, CHECK FOR OVERFLOW
*           ***
30TOFLO2  =E0+1;  (AC)=AC;  IF¬ALU(59)/ALU(48) THEN 30NOOFLO
          AC%MQ=SHIFT(AC%MQ,A1);  E0=E0+1
*           ***
*           *  AC:MQ NOW CONTAINS THE 96 BIT SUM
*           ***
30NOOFLO  NEWPARCEL; IF OPCODE(2) THEN 30RND
*           ***
*           *  EPILOG FOR DOUBLE PRECISION ADD: SHIFT LOWER SUM RIGHT
*           *         12 BITS, REDUCE EXPONENT BY 60(OCTAL).
*           ***
          AC=MQ;  MQ=0;  IF ¬AC(59) THEN 30SUMPOS
          MQ=-0       * USED WHEN SUM IS NEGATIVE.
30SUMPOS  AC=SHIFT(AC%MQ,R4);  =E0-60
          AC=SHIFT(AC%MQ,R4);  E0=E0-60;  IF FOFL THEN FLRESFLO
          AC=SHIFT(AC%MQ,R4);  GO WXIFLOAT
*           ***
*           *  EPILOG FOR ROUNDED ADD.
*           *  30RNDFLO :THIS INSTRUCTION WILL BE REACHED FROM THE FOLLOWING
*           *             INSTRUCTIONS WHEN OVERFLOW OCCURS.
*           ***
30RNDFLO  AC%MQ=SHIFT(AC%MQ,A1);  E0=E0+1;  GO WXIFLOAT
*                                 * CAN OVERFLOW OCCUR MORE THAN ONCE ?,IF SO, MUST
*                                 * TEST FOR EXPONENT OUT OF RANGE HERE.
30RND     IF AC(59) THEN 30RNDNEG
          (AC)=AC+0[NOP]; X1=E0%AC; IF ¬MQ(59) THEN NEWINSTR
          AC=AC+0[NOP];  =E0+1;  IF ALU(59)/ALU(48) THEN 30RNDFLO ELSE WXIFLOAT
*           ***
*           *  30RNDNEG: IN ROUNDING A NEGATIVE SUM, 1 IS SUBTRACTED FROM
*           *             AC WHEN MQ(59)=0.
*           ***
30RNDNEG  (AC)=AC-0[G]; X1=E0%AC; IF MQ(59) THEN NEWINSTR
          AC=AC-0[G];  =E0+1;  IF ALU(59)/ALU(48) THEN 30RNDFLO ELSE WXIFLOAT
*
*************
*
* FXI XJ-XK - OP 31
*
*************
31        E0%BUF=XK
31A       AC=-BUF;  E1%BUF=XJ;  IF ILL(E0) THEN 30ILLEXP ELSE 30B
*
*************
*
* DXI XJ+XK - OP 32
```

```
*
**************
*
32        EO%BUF=XK;  GO 30A
*
**************
*
* DXI XJ-XK - OP 33
*
**************
*
33        EO%BUF=XK; GO 31A
*
**************
*
* RXI XJ+XK - OP 34
*
**************
*
34        EO%BUF=XK;  GO 30A
*
**************
*
* RXI XJ-XK - OP 35
*
**************
*
35        EO%BUF=XK; GO 31A
*
**************
*
* IXI XJ+XK - OP 36- INTEGER SUM OF XJ AND XK TO XI.
*             (5 CYCLES)
*
**************
*
36        BUF=XK
          AC=BUF;  BUF=XJ
LONGADD   =AC+BUF; NEWPARCEL
          AC=AC+BUF; GO WXI
*
**************
*
* IXI XJ-XK - OP 37- INTEGER DIFFERENCE OF XJ AND XK TO XI.
*             (5 CYCLES)
*
**************
*
37        BUF=XK
          AC=-BUF;  BUF=XJ;  GO LONGADD
*
*****************************************************************************
*                                                                           *
* FLOATING POINT MULTIPLICATION OF XJ AND XK                                *
```

```
*                                                                      *
*    MULTIPLICATION (OF POSITIVE NUMBERS) IS PERFORMED BY FIRST COMPUTING      *
* XJ, 2*XJ,...., 7*XJ, 8*XJ, STORING THEM IN Y1, Y2,...., Y7 AND Y0,RESPECTIVELY,*
* AND THEN PROCESSING 4 BITS OF XK AT A TIME.                          *
*                                                                      *
*    WHEN THE PROCESSING BEGINS AC:MQ = 0:XK , AND AFTER EACH CYCLE, AC:MQ WILL*
* CONTAIN BOTH THE PARTIAL RESULT OF THE MULTIPLICATION IN THE UPPER BITS , AND*
* WHAT IS LEFT TO PROCESS OF XK IN THE LOWER BITS.                     *
*                                                                      *
*    WE DEFINE   M  AS    THE RIGHMOST 4 BITS OUT OF MQ, I.E. 0 ≤ M ≤ 15.  *
*                                                                      *
*    IN EACH CYCLE, AC:MQ IS SHIFTED 4(ARITHMETIC) RIGHT, SO WHEN ADDING TO    *
* THE AC THE PROPER MULTIPLE OF XJ (IN BUF), ADDITION IS PROPERLY ALIGNED.     *
*                                                                      *
*    BASICALLY THE ALGORITHM IS AS FOLLOWS:                            *
* IF 0 ≤ M ≤ 8 ,ONE ADDS  BUF = M *(XJ) TO AC.                         *
* IF 9 ≤ M ≤ 15, ONE SUBTRACTS  BUF =(16 - M)*(XJ) FROM AC AND ADDS AN EXTRA   *
*                XJ IN THE NEXT CYCLE (WHICH IS EQUIVALENT TO ADDING 16*(XJ)).*
*                                                                      *
*    THE ACTUAL COMPUTATION HAS TO TAKE INTO ACCOUNT WHETHER THE PREVIOUS      *
* CYCLE WAS AN ADD OR A SUBTRACT. THE NOTATION 4OA, 4OS, 4OAA, ETC., REPRESENTS*
* THE DIFFERENT POSIBILITIES ACCORDING TO AN ADD OR A SUBTRACT CYCLE. FOR      *
* EXAMPLE:4OAS INDICATES PREVIOUS CYCLE IS ADD AND PRESENT CYCLE SUBTRACT.     *
*                                                                      *
*    TABLE FOR THE SELECTION FUNCTION                                  *
*                                                                      *
*      PREVIOUS          VALUE OF      SET BUF =        PRESENT       *
*       CYCLE              M           (YM OR YN)        CYCLE        *
*                  **                                                  *
*                 *  0 ≤ M ≤ 8        M*(XJ)             ADD          *
*        ADD    ****                                                   *
*                 *  8 < M ≤ 15     (16-M)*(XJ)          SUBTR        *
*                  **                                                  *
*                  **                                                  *
*                 *  0 ≤ M ≤ 7       (M+1)*(XJ)          ADD          *
*                 *                                                   *
*      SUBTR    ****  7 < M ≤ 14   (16-(M+1))*(XJ)      SUBTR        *
*                 *                                                   *
*                 *      M=15            0              SUBTR        *
*                  **                                                  *
*                                                                      *
*    THE SELECTION OF BUF= YM OR YN IS DONE FROM E REGISTER AS FOLLOWS:        *
*  . MM ARE 8 BITS FORMED BY    SETTING BITS 0-3 EQUAL TO M                  *
*                               BITS 4-7 EQUAL TO THE 1 COMPLEMENT OF M      *
*  . YM MEANS: IF E1(7) IS SET FETCH Y REGISTER WHOSE NUMBER IS GIVEN BY      *
*       BITS 0-2 OF E1, ELSE FETCH 0                                   *
*  . YN MEANS: IF E1(3) IS SET FETCH Y REGISTER WHOSE NUMBER IS GIVEN BY      *
*       BITS 4-6 OF E1, ELSE FETCH 0                                   *
*  . [F] INDICATES THAT THE ADDITION IN E1 IS DONE AS IF E1 CONSISTED OF 3    *
*    SEPARATE WORDS OF 4 BITS EACH WITH NO CARRY AND NO PROPAGATE.     *
*                                                                      *
*    ADDITION AC+BUF[G] OR SUBTRACTION AC-BUF[NOP] IS DONE IN 2-COMPLEMENT     *
* REPRESENTATION TO AVOID THE PROBLEM OF THE END AROUND CARRY.         *
*                                                                      *
```

```
*    THERE ARE THREE TYPES OF FLOATING MULTIPLICATION, FLOATING, ROUND          *
* FLOATING AND DOUBLE PRECISION FLUATING.                                       *
*                                                                               *
*********************************************************************************
*
**************
*
* FXI XJ*XK -OP 40 -FLOATING PRODUCT OF XJ AND XK TO XI.
*
**************
40       E1%BUF=XJ;  MQ=0;  IF REG(59) THEN 40XJNEG ELSE 40XJPOS
40XJPOS  AC=BUF;  E2%BUF=XK;  IF ILL(E1) THEN 40ILLEXP ELSE 40FORMMP
40XJNEG  AC=-BUF;  E2%BUF=XK;  IF ILL(E1) THEN 40ILLEXP
*          ***
*          *  AT THIS POINT AC=ABS(XJ),BUF=XK, E1= EXP(XJ), E2=EXP(XK).
*          *  AND WE PROCEED TO SET Y1=ABS(XJ),Y2= 2*ABS(XJ),...,Y7= 7*ABS(XJ),
*          *  Y0 = 8*ABS(XJ).
*          ***
40FORMMP Y1=AC;  AC=SHIFT(AC%MQ,L1);  IF ILL(E2) THEN 40ILLEXP
         Y2=AC;  AC=SHIFT(AC%MQ,L1);  IF BUF(59) THEN 40XKNEG
         BUF=Y1;  MQ=BUF;  GO 40B
40XKNEG  BUF=Y1;  MQ=-BUF
40B      Y4=AC;  AC=SHIFT(AC%MQ,L1)
         Y0=AC;  =AC-BUF
         BUF=Y2;  AC=AC-BUF
         Y7=AC;  =AC-BUF
         AC=AC-BUF;  IF ZERO(E1) THEN 40XJZERO
         Y5=AC;  =AC-BUF;  IF ZERO(E2) THEN WXIZERON
         AC=AC-BUF;  =E1+E2
         Y3=AC;  AC=SHIFT(AC%MQ,L1);  E0=E1+E2;  IF XFOFL THEN FLRSFLON
40INTMUL Y6=AC;  AC=0;  E2=15
*          ***
*          *  AT THIS POINT, Y REGISTERS CONTAIN APPROPRIATE MULTIPLES OF
*          *  ABS(XJ), MQ = ABS(XK), AC= 0 , E0 = EXPONENT OF THE PRODUCT.
*          *  E2 = 15 (OCTAL) HAS BEEN INITIALIZED TO COUNT THE NUMBER OF
*          *  CYCLES REQUIRED TO PROCESS XK.
*          *
*          *  START MAIN MULTIPLY LOOP.
*          *
*          ***
*                START MAIN MULTIPLY LOOP
         AC%MQ=SHIFT(AC%MQ,A4);  E1=MM+17[F];  IF M>8 THEN 40S
         AC%MQ=SHIFT(AC%MQ,A4);BUF=YM;E1=MM+ 17[F];IF M>6 THEN 40AS ELSE 40AA
40S      AC%MQ=SHIFT(AC%MQ,A4);BUF=YN;E1=MM+360[F];IF M>7 THEN 40SS ELSE 40SA
40AA     =AC+BUF[G];  E2=E2-1[F];  IF ¬EALU(0-3) THEN 40ADONE
         AC%MQ=SHIFT(AC+BUF%MQ,A4)[G];     BUF=YM;
+                             E1=MM+ 17[F];  IF M>8 THEN 40AS ELSE 40AA
40AS     =AC+BUF[G];  E2=E2-1[F];  IF ¬EALU(0-3) THEN 40ADONE
         AC%MQ=SHIFT(AC+BUF%MQ,A4)[G];     BUF=YN;
+                             E1=MM+360[F];  IF M>7 THEN 40SS ELSE 40SA
40SA     =AC-BUF[NOP];  E2=E2-1[F]
         AC%MQ=SHIFT(AC-BUF%MQ,A4)[NOP];     BUF=YM;
+                             E1=MM+ 17[F];  IF M>8 THEN 40AS ELSE 40AA
40SS     =AC-BUF[NOP];  E2=E2-1[F]
```

```
        AC%MQ=SHIFT(AC-BUF%MQ,A4)[NOP];      BUF=YN;
+                           E1=MM+360[F];  IF M>7 THEN 40SS ELSE 40SA
*         ***
*         *  40ADONE:  MULTIPLICATION OF COEFFICIENTS IS COMPLETED ON THIS
*                      CYCLE.  PRODUCT IS IN LOW-ORDER 48 BITS OF AC AND
*                      HIGH-ORDER 48 BITS OF MQ.
*                      .IF ALU(47)=1 PRODUCT  IS ALREADY NORMALIZED,
*                                    GO TO 40NOSHFT
*                      .IF ALU(47)=0 PRODUCT IS NOT NORMALIZED.  IT SHOULD
*                                    BE NORMALIZED (BY A 1 BIT LEFT SHIFT)
*                                    ONLY IF BOTH OPERANDS WERE NORMALIZED.
*         ***
40ADONE   AC=AC+BUF[G];  BUF=XJ;  IF ALU(47) THEN 40NOSHFT
        BUF=XK;  =BUF;  IF ¬ALU(59)/ALU(47) THEN 40NOSHFT
*                           *    ¬(ALU(59)/ALU(47)) CHECKS WHETHER BUF =XJ
*                           * IS NORMALIZED.
        =BUF;  =E0-1;  IF ¬ALU(59)/ALU(47) THEN 40NOSHFT
*                           * ¬(ALU(59)/ALU(47)) CHECKS WHETHER BUF = XK IS
*                           * NORMALIZED.
        AC%MQ=SHIFT(AC%MQ,L1);  E0=E0-1; IF OPCODE(1) THEN 40DP
40NOSHFT IF OPCODE(1) THEN 40DP; =E0+60
        IF OPCODE(0) THEN 40ROUND; E0=E0+60;  BUF=XJ
*         ***
*         *  40 SETSGN: NEXT TWO INSTRUCTIONS SET AC TO SIGNED PRODUCT.
*         *             ON ENTRY :   BUF(59)=SIGN(XJ)
*         *                          AC= ABS(PRODUCT).
*         ***
40SETSGN BUF=XK;  IF ¬REG(59)/BUF(59) THEN 40WX1
40SETNEG AC=-AC; NEWPARCEL; IF FOFL(E0) THEN FLRESFLO ELSE WX1FLOAT
40WX1     XI=E0%AC; NEWPARCEL;  IF FOFL(E0) THEN FLRESFLO ELSE NEWINSTR
*         ***
*         *  FOR DP AND ROUND MULTIPLY
*         ***
40ROUND   (AC)=AC+0[NOP];  IF ¬MQ(59) THEN 40SETSGN
*                           * IF MQ(59)=1,TO ROUND THE RESULT WE ADD 1 TO AC
        AC=AC+0[NOP];  =E0+1;  IF ¬ALU(59)/ALU(48) THEN 40SETSGN
        AC=SHIFT(AC%MQ,A1);  E0=E0+1;  BUF=XK;
+       IF REG(59)/BUF(59) THEN 40SETNEG ELSE 40WX1
*         ***
*         *  40DP: INSTRUCTIONS ARE SIMILAR TO THOSE OF DP ADD.
*         ***
40DP      MQ=0;  AC=MQ
        AC%MQ=SHIFT(AC%MQ,R4)
        AC%MQ=SHIFT(AC%MQ,R4); BUF=XJ
        AC%MQ=SHIFT(AC%MQ,R4); BUF=XK;
+       IF REG(59)/BUF(59) THEN 40SETNEG ELSE 40WX1
*         ***
*         *  SPECIAL RESULTS FOR ZERO AND ILLEGAL EXPONENTS.
*         *  ON ENTRY E1= EXP(XJ), E2= EXP(XK).
*         *  ALGORITHM:
*         *
*         *       IF (XJ = INDEF) GO TO INDEFOP
*         *       IF (XK = INDEF) GO TO INDEFOP
*         *     *    IF WE GET HERE EITHER XJ OR XK (OR BOTH) ARE INFINITE
```

```
*           *       IF (XJ = INF )  GO TO XJINF
*           *     *    CHECK FOR ZERO BECAUSE XK IS INFINITY
*           *       IF (XJ = 0 ) GO TO WXIINDEF
*           *       AC = SIGN(XK/XJ)
*           *       GO TO WXIINF
*           * XJINF IF (XK = 0 ) GO TO WXIINDEF
*           *       AC = SIGN(XK/XJ)
*           *       GO TO WXIINF     *XI=INFINITY, SIGN GIVEN BY AC.
*           ***
40ILLEXP BUF=XJ;  IF INDEF(E1) THEN INDEFOPN
         AC=BUF;  BUF=XK;  IF INDEF(E2) THEN INDEFOPN
         AC=AC/BUF;  E(=2002;  IF MODE2 THEN ERROR
         IF INF(E1) THEN 40XJINF
         NEWPARCEL; IF ZERO(E1) THEN WXIINDEF ELSE WXIINF
40XJINF  NEWPARCEL; IF ZERO(E2) THEN WXIINDEF ELSE WXIINF
40XJZERO Y5=AC;  =AC-BUF;  IF ¬ZERO(E2) THEN WXIZERON
         AC=AC-BUF; IF ¬OPCODE(1) THEN WXIZERON
         Y3=AC;  AC=SHIFT(AC%MQ,L1);  E0=6000;  GO 40INTMUL
*
**************
*
* RXI XJ*XK - OP 41-ROUND FLOATING PRODUCT OF XJ AND XK TO XI
*
**************
*
41       E1%BUF=XJ;  MQ=0;  IF NEG(59) THEN 40XJNEG ELSE 40XJPOS
*
**************
*
* DXI XJ*XK - OP 42. FLOATING DOUBLE PRECISION PRODUCT OF XJ AND XK TO XI
*
**************
*
42       E1%BUF=XJ;  MQ=0;  IF NEG(59) THEN 40XJNEG ELSE 40XJPOS
*
**************
*
* MXI JK - OP 43 - FORM MASK IN XI, JK BITS
*                 -INSTRUCTION SETS HIGH ORDER JK BITS OF XI TO ONE.
*                 -FIRST THREE INSTRUCTIONS SET HIGH ORDER BIT OF AC AND SO,
*                  AN ARITHMETIC RIGHT SHIFT (OP 21) JK-1 PLACES,
*                  GIVES THE DESIRED RESULT.
**************
*
43       AC=0;  =0-JK
         AC%MQ=SHIFT(AC%MQ,01);  E0=0-JK; NEWPARCEL; IF ¬EALU(11) THEN WXIZERO
         AC%MQ=SHIFT(AC%MQ,R1); =7776-E0
         E0=7776-E0; IF EALU(4)∨EALU(5) THEN 21SH16 ELSE 21TRY4
*
*****************************************************************************
*                                                                           *
* FLOATING POINT DIVISION                                                   *
*                                                                           *
*    TO DIVIDE XJ BY XK (BOTH POSITIVE NUMBERS) FIRST XJ IS PLACED IN AC AND *
```

```
* XK IN THE BUF. THE 50 BIT QUOTIENT WILL APPEAR IN MQ.                          *
*                                                                               *
*    IF WE DENOTE BY AC(HIGH) AND BUF(HIGH) BITS 45-48 OF AC AND BUF, RESPEC-   *
* TIVELY, THE DIVIDE ALGORITHM FOLLOWS:                                         *
*                                                                               *
*                AC = XJ                                                        *
*                BUF= XK                                                        *
*                MQ = 0          * CARRIES QUOTIENT                             *
*                SHC= 0          * INITIALIZES SHIFT COUNT                      *
*        LOOP1   AC = AC * 2                                                    *
*                MQ = MQ * 2                                                    *
*        LOOP12  SHC= SHC + 1                                                   *
*                IF (SHC = 50) GO TO DONE                                       *
*                IF ( AC(HIGH) < BUF(HIGH)) GO TO LOOP1                         *
*                AC = AC - BUF                                                  *
*                IF (AC < 0 )  GO TO READD  * AC WAS SMALLER THAN BUF AND       *
*                                           * SUBTRACTION WAS NOT INDICATED     *
*                 AC = AC * 2                                                   *
*                 MQ =(MQ * 2) + 1          * RECORDS QUOTIENT                  *
*                 GO TO LOOP12                                                  *
*        READD    AC = AC + BUF                                                 *
*                 GO TO LOOP1                                                   *
*        DONE     END                                                           *
*                                                                               *
*    TO IMPLEMENT THIS ALGORITHM IN THE MACHINE WE PROCEED AS FOLLOWS:          *
*  1.AC =(AC * 2) AND  MQ = (MQ *2) IS DONE BY AC:MQ=SHIFT(AC:MQ,Z1)            *
*                                                                               *
*  2.AC =(AC * 2) AND  MQ = (MQ *2)+1 IS DONE BY AC:MQ=SHIFT(AC:MQ,O1)          *
*                                                                               *
*  3.MQ WILL CARRY BOTH, THE QUOTIENT AND THE SHIFT COUNT,NAMELY,BEFORE         *
*    STARTING DIVISION PROCESS, A 1 BIT IS FORCED INTO BIT 0 OF MQ, SO THAT     *
*    SHC = 50 IS EQUIVALENT TO MQ(49)=1.                                        *
*                                                                               *
*  4.THE TEST AC << BUF COMPARES BITS 44-47 OF AC AGAINST BITS 45-48 OF BUF     *
*    BECAUSE TEST IS CHECKING A CONDITION EXISTING PRIOR TO SHIFTING            *
*                                                                               *
*  5.EO WILL CARRY THE EXPONENT OF THE QUOTIENT, I.E. THE DIFFERENCE OF THE     *
*    EXPONENTS MINUS 60(OCTAL) (BECAUSE MINUEND WAS SHIFTED 48 BITS LEFT).      *
*                                                                               *
*********************************************************************************
*
**************
*
* FXI XJ/XK - OP 44- FLOATING DIVIDE XJ BY XK TO XI
*
**************
*
*          ***
*          *  SET YO= ABS(XK), SIGN(Y1)= SIGN OF RESULT, AC= ABS(XJ).
*          ***
44       E2%BUF=XK;  IF REG(59) THEN 44XKNEG
44XKPOS  AC=BUF;  E1%BUF=XJ;  IF ILL(E2) THEN 441LLEXP
         YO=AC;  AC=BUF;  IF ILL(E1) THEN 44ILLEXP ELSE 44A
*          ***
```

```
*           *  44XKNEG: WHEN XK <0,SETTING AC=Y1= -XJ MAKES Y1 HAVE SIGN
*           *           OF RESULT.
*           ***
44XKNEG   AC=-BUF;  E1%BUF=XJ;  IF ILL(E2) THEN 44ILLEXP
          Y0=AC;  AC=-BUF;  IF ILL(E1) THEN 44ILLEXP
44A       Y1=AC;  MQ=0;  IF ZERO(E1) THEN 44XJZERO
          BUF=Y0;  IF ZERO(E2) THEN WXIINFN
*                             * IF ZERO(E2) IS TRUE, XJ IS FINITE, XK=0,
*                             * QUOTIENT IS INFINITE.
          =E1-E2;  IF AC(59) THEN 44COMP
          E0=E1-E2;  IF XFOFL THEN FLRSFLON ELSE 44B
44COMP    E0=E1-E2;  AC=-AC;  IF XFOFL THEN FLRSFLON
44B       =E0-60;  AC%MQ=SHIFT(AC%MQ,A1)
*           ***
*           *  NEXT TWO INSTRUCTIONS FORCE A 1 BIT IN ZERO BIT OF MQ TO START
*           *  SHIFT COUNT.
*           ***
          E0=E0-60;  AC%MQ=SHIFT(AC%MQ,01);
+         IF AC<<BUF^¬MQ(49) THEN 44LOOP ELSE 44SUBTR
44LOOP    AC%MQ=SHIFT(AC%MQ,Z1);  IF AC<<BUF^¬MQ(49) THEN 44LOOP
44SUBTR   =AC-BUF; IF MQ(50) THEN 44DONE
          AC=AC-BUF; IF ALU(59) THEN 44READD
          AC%MQ=SHIFT(AC%MQ,01); IF AC<<BUF^¬MQ(49) THEN 44LOOP ELSE 44SUBTR
44READD   =AC+BUF
          AC=AC+BUF; GO 44LOOP
*                  AT THIS POINT MQ(0-49) HAVE A 50-BIT QUOTIENT, MQ(51)=1
44DONE    AC=MQ;  BUF=Y1;  IF MQ(49) THEN 44SHIFT
          (AC)=AC+0[NOP];  IF OPCODE(0) THEN 44ROUND
*           ***
*           *  44NORND: WE GET HERE IF M(49)=0. SHIFT ARITHMETIC 1 RIGHT
*           *           GIVES QUOTIENT.
*           ***
44NORND   NEWPARCEL; AC=SHIFT(AC%MQ,A1);  IF BUF(59) THEN 44NEGRES
          XI=E0%AC;  IF FOFL(E0) THEN FLRESFLO ELSE NEWINSTR
44NEGRES  AC=-AC;  IF FOFL(E0) THEN FLRESFLO ELSE WXIFLOAT
*           ***
*           *  44SHIFT: WHEN MQ(49)=1,SINCE MQ CONTAINS A 50 BIT QUOTIENT,
*           *           2 SHIFTS RIGHT ARE REQUIRED TO OBTAIN ANSWER.
*           ***
44SHIFT   =E0+1;  AC=SHIFT(AC%MQ,A1)
          (AC)=AC+0[NOP];  E0=E0+1;  IF OPCODE(0) THEN 44ROUND ELSE 44NORND
*           ***
*           *  44ROUND: ADD 1 TO LOWER BIT AND CHECK FOR OVERFLOW.
*           ***
44ROUND   AC=AC+0[NOP];  GO 44NORND
*           ***
*           *  44XJZERO: COMES FROM LINE 44A WHEN NUMERATOR IS ZERO.
*           ***
44XJZERO  NEWPARCEL; IF ZERO(E2) THEN WXIINDEF ELSE WXIZERO
*           ***
*           *  AT THIS POINT ONE OR BOTH E1 AND E2 ARE INFINITE.
*           ***
44ILLEXP  BUF=XJ;  IF INDEF(E1) THEN INDEFOPN
          AC=BUF;  BUF=XK; NEWPARCEL;  IF INDEF(E2) THEN INDEFOP
```

```
        AC=AC/BUF;  E0=2002;  IF MODE2 THEN ERROR
        IF ¬INF(E1) THEN WXIZERO
        IF INF(E2) THEN WXIINDEF ELSE WXIINF
*
***************
*
* RXI XJ/XK . OP 45. ROUND FLOATING DIVIDE XJ BY XK TO XI.
*
***************
*
45      E2%BUF=XK;  IF REG(59) THEN 44XKNEG ELSE 44XKPOS
*
***************
*
* NO . OP 46  . NO OPERATION
*
***************
*
46      NEWPARCEL; IF ICHECK THEN ICHECK ELSE NEWINSTR
*
***************
*
* CXI XK - OP 47 - COUNT OF THE NUMBER OF 1 BITS IN XK TO XI
*                  .BC COUNTS THE NUMBER OF 1 BITS IN THE LOW 4 BITS OF AC
*                  .IF XK IS POSITIVE, LOAD XK INTO AC AND ACCUMULATE IN E0
*                   A COUNT OF THE 1 BITS IN AC
*                  .IF XK IS NEGATIVE, LOAD -XK INTO AC AND ACCUMULATE IN E0
*                   60 - COUNT OF 1 BITS IN AC = 60 - COUNT OF 0 BITS IN XI
*                   = COUNT OF 1 BITS IN XI
*
***************
*
47      BUF=XK; MQ=0; NEWPARCEL; IF REG(59) THEN 47XKNEG
        AC=BUF; E0=0
47LOOP  =E0+BC; IF AC=0 THEN 47END
        E0=E0+BC; AC=SHIFT(AC%MQ,R4); GO 47LOOP
47END   AC=E0; GO WXI
47XKNEG AC=-BUF; E0=74
47NLOOP =E0-BC; IF AC=0 THEN 47END
        E0=E0-BC; AC=SHIFT(AC%MQ,R4); GO 47NLOOP
*****************************************************************************
*                                                                           *
* SAI GROUP -OP 50 - SAI AJ+K  SET AI TO AJ+K                               *
*               51 - SAI BJ+K  SET AI TO BJ+K                               *
*               52 - SAI XJ+K  SET AI TO XJ+K                               *
*               53 - SAI XJ+BK SET AI TO XJ+BK                              *
*               54 - SAI AJ+BK SET AI TO AJ+BK                              *
*               55 - SAI AJ-BK SET AI TO AJ-BK                              *
*               56 - SAI BJ+BK SET AI TO BJ+BK                              *
*               57 - SAI BJ-BK SET AI TO BJ-BK                              *
*                                                                           *
*         INSTRUCTIONS SET AI TO THE SPECIFIED ADDRESS-                     *
*         SETTING AI THROUGH A5 TO AN ADDRESS LOADS CONTENTS OF THAT        *
*               LOCATION INTO ASSOCIATED X REGISTER.                        *
```

```
*          SETTING A6 OR A7 STORES CONTENTS OF THE X REGISTER AT THE SPECIFIED  *
*               LOCATION                                                        *
*          SETTING A0 CAUSES NO MEMORY REFERENCE                                *
*                                                                               *
*********************************************************************************
*
50        BUF=AJ;  AC=K;     NEWPARCEL
SAADD     =AC+BUF[18];  NEWPARCEL;  IF I=0 THEN NOLOAD
          AC=AC+BUF[18];  IF ¬NIWEMPTY THEN TESTI
*           IF RNI IS NOT COMPLETE, WAIT FOR CMDONE AND THEN LOAD NIW
LDWAIT1   NIW=CMRD; IF ¬CMDONE THEN LDWAIT1
TESTI     CLEAR; A1=AC;  IF I>5 THEN STORE
*           LOAD SEQUENCE%
*              WAIT FOR CMDONE TO DROP
LOAD      IF CMDONE THEN LOAD
*              ISSUE READ REQUEST
          MA=AC;  READ; IF LASTPARCEL THEN LDLASTP
*              WAIT FOR CM FETCH TO COMPLETE
LDWAIT2   AC=CMRD;  IF ¬CMDONE THEN LDWAIT2
*              STORE FETCHED DATA IN XI
          XI=AC;  CLEAR;  LATCH I; GO OPCODEBRANCH
LDLASTP   AC=CMRD; IF ¬CMDONE THEN LDLASTP
          XI=AC; CLEAR; P=P+1; GO LOADCIW
*           STORE SEQUENCE%
*              GET WORD TO STORE (XI);  WAIT FOR CMDONE TO DROP
STORE     BUF=XI;  IF CMDONE THEN STORE
*              ISSUE WRITE REQUEST
          AC=BUF;  MA=AC;  WRITE;  IF LASTPARCEL THEN STLASTP
*              WAIT FOR CM TO ACCEPT DATA
STWAIT    IF ¬CMDONE THEN STWAIT
          CLEAR; LATCH I; GO OPCODEBRANCH
STLASTP   IF ¬CMDONE THEN STLASTP
          CLEAR; P=P+1; GO LOADCIW
*           I=0 CASE%  NO LOAD OR STORE
NOLOAD    AC=AC+BUF[18]
          AI=AC;  LATCH I; IF ICHECK THEN ICHECK ELSE OPCODEBRANCH
51        BUF=BJ;  AC=K;     NEWPARCEL;  GO SAADD
52        BUF=XJ;  AC=K;     NEWPARCEL;  GO SAADD
53        BUF=BK
          AC=BUF;  BUF=XJ;  GO SAADD
54        BUF=BK
          AC=BUF;  BUF=AJ;  GO SAADD
55        BUF=BK
          AC=-BUF;  BUF=AJ;  GO SAADD
56        BUF=BK
          AC=BUF;  BUF=BJ;  GO SAADD
57        BUF=BK
          AC=-BUF;  BUF=BJ;  GO SAADL
*
*********************************************************************************
*                                                                               *
* SBI GROUP -OP 60 - SBI AJ+K  SET B1 TO AJ+K                                   *
*               61 - SBI BJ+K  SET B1 TO BJ+K                                   *
*               62 - SBI XJ+K  SET B1 TO XJ+K                                   *
```

```
*               63 - Sbi XJ+BK SET Bi TO XJ+BK                                  *
*               64 - SBI AJ+BK SET BI TO AJ+BK                                  *
*               65 - Sbi AJ-BK SET Bi TO AJ-BK                                  *
*               66 - Sbi BJ+BK SET Bi TO BJ+BK                                  *
*               67 - Sbi BJ-BK SET Bi TO BJ-BK                                  *
*                                                                               *
*         INSTRUCTIONS SET Bi TO SPECIFIED ADDRESS.                             *
*         B0 IS PERMANENTLY SET TO ZERO                                         *
*                                                                               *
*********************************************************************************
*
60        BUF=AJ;  AC=K; NEWPARCEL
SBADD     =AC+BUF[18]; NEWPARCEL;  IF I=0 THEN NEWINSTR
          AC=AC+BUF[18]
          Bi=AC;  LATCH 1; IF ICHECK THEN ICHECK ELSE OPCODEBRANCH
61        BUF=BJ;  AC=K; NEWPARCEL;  GO SBADD
62        BUF=XJ;  AC=K; NEWPARCEL;  GO SBADD
63        BUF=BK; IF I=0 THEN NEWINSLO
          AC=BUF;  BUF=XJ;  GO SBADD
64        BUF=BK; IF I=0 THEN NEWINSLO
          AC=BUF;  BUF=AJ;  GO SBADD
65        BUF=BK; IF I=0 THEN NEWINSLO
          AC=-BUF;  BUF=AJ;  GO SBADD
66        BUF=BK; IF I=0 THEN NEWINSLO
          AC=BUF;  BUF=BJ;  GO SBADD
67        BUF=BK; IF I=0 THEN NEWINSLO
          AC=-BUF;  BUF=BJ;  GO SBADD
*
*********************************************************************************
*                                                                               *
* SXi GROUP -OP 70 - SXi AJ+K  SET Xi TO AJ+K                                   *
*               71 - SXi BJ+K  SET Xi TO BJ+K                                   *
*               72 - SXi XJ+K  SET Xi TO XJ+K                                   *
*               73 - SXi XJ+BK SET Xi TO XJ+BK                                  *
*               74 - SXi AJ+BK SET Xi TO AJ+BK                                  *
*               75 - SXi AJ-BK SET Xi TO AJ-BK                                  *
*               76 - SXi BJ+BK SET Xi TO BJ+BK                                  *
*               77 - SXi BJ-BK SET Xi TO BJ-BK                                  *
*                                                                               *
*         INSTRUCTIONS SET Xi TO SPECIFIED ADDRESS.                             *
*         IN ALL CASES, AN 18-BIT ONE'S-COMPLEMENT ADDITION IS PERFORMED        *
*         AND THE SIGN BIT IS THEN EXTENDED TO THE HIGH-ORDER 42 BITS TO        *
*         FORM A 60-BIT SUM.                                                    *
*                                                                               *
*********************************************************************************
*
70        BUF=AJ; AC=K;    NEWPARCEL
SXADD     =AC+BUF[18]
          AC=AC+BUF[18];  NEWPARCEL; GO WXI
71        BUF=BJ;  AC=K;    NEWPARCEL;  GO SXADD
72        BUF=XJ;  AC=K;    NEWPARCEL;  GO SXADD
73        BUF=BK
          AC=BUF;  BUF=XJ;  GO SXADD
74        BUF=BK
```

```
        AC=BUF;  BUF=AJ;  GO SXADD
75      BUF=BK
        AC=-BUF;  BUF=AJ;  GO SXADD
76      BUF=BK
        AC=BUF;  BUF=BJ;  GO SXADD
77      BUF=BK
        AC=-BUF;  BUF=BJ;  GO SXADD
*
```

This report was prepared as an account of Government sponsored work. Neither the United States, nor the Administration, nor any person acting on behalf of the Administration:

A. Makes any warranty or representation, express or implied, with respect to the accuracy, completeness, or usefulness of the information contained in this report, or that the use of any information, apparatus, method, or process disclosed in this report may not infringe privately owned rights; or

B. Assumes any liabilities with respect to the use of, or for damages resulting from the use of any information, apparatus, method, or process disclosed in this report.

As used in the above, "person acting on behalf of the Administration" includes any employee or contractor of the Administration, or employee of such contractor, to the extent that such employee or contractor of the Administration, or employee of such contractor prepares, disseminates, or provides access to, any information pursuant to his employment or contract with the Administration, or his employment with such contractor.

GAYLORD 142			PRINTED IN U S A

N.Y.U. Courant Institute of
Mathematical Sciences
251 Mercer St.
New York, N. Y. 10012